ESD-TR-91-225

MTR-11158

Ada and X Window System Integration

By

C. M. Byrnes

January 1992

Prepared for

Director, Systems & Software
Design Center of Excellence
Electronic Systems Division
Air Force Systems Command
United States Air Force
Hanscom Air Force Base, Massachusetts

Project No. 5800

Prepared by

The MITRE Corporation
Bedford, Massachusetts

Contract No. F19628-89-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

JAMES HENSLEE, GS-14
Chief Engineer

ANDREW HODYKE, GM-13
STARS Deputy Program Manager

FOR THE COMMANDER

ROBERT J. KENT, GM-15
Director, Systems & Software
Design Center of Excellence

EXECUTIVE SUMMARY

MITRE's role in the Software Technology for Adaptable and Reliable Systems (STARS) Ada/X joint development effort was to serve as technical contributor and consensus builder between the two development teams at Scientific Applications International Corporation (SAIC, subcontractor to Boeing for Ada/X) and Unisys. Besides working with the two developers to come up with timely and technically sound compromises for their initial independent Ada/X approaches to develop a common binding, we also tried to view the resultant windowing system from the perspective of both the eventual Ada applications developer and the customers who must eventually maintain it. MITRE attendees at various Ada/X Technical Interchange Meetings (TIMs) tried to capture as many of the assumptions, decisions, and rationales as possible in the realization that the user community may need to know this information before trying these bindings on applications.

One area that has to be considered when developing an application using Ada/X is the software design method(s) being used. The X Window System™[1] assumes an Object-Oriented Programming (OOP) approach that is implemented manually in the default C programming language (Scheifler, 1988). Ada has its own built-in OOP constructs of which Object-Oriented Designs (OOD) often try to take advantage. The STARS Ada/X work tried to straddle the boundary between following what the reference X implementations in C provided (C/X, with which most existing X programmers will be most familiar) and what an idealized Ada/X binding would provide. This paper provides an overview into some of the OOD and OOP issues that programmers and customers have to consider when using Ada and X.

Ada provides a variety of OOP constructs to choose from, and it turns out that SAIC and Unisys made different choices in their individual implementations of Ada/X. While this complicated the TIMs' task of developing an Ada/X binding that abstracted the different implementation details away from the application programmer, there may be some advantages in having different OOP implementation approaches. Experience with different Ada compilers has shown that individual compilers do a good job, efficiently handling some Ada constructs but not others. There are variants across compiler vendors, so one compiler might do a good job with one approach but not another. The different Ada/X implementation approaches allow application developers to choose which of the two bindings best matches the strengths of their compiler. But this does require detailed knowledge by the application development organization as to the specific strengths and weakness of their chosen compiler(s).

The STARS program has redirected its efforts towards more commercial acceptance of whatever tools and products come out of the three STARS prime contractors (Boeing, IBM, and Unisys). This is due partially to the recognition that government-funded programs cannot be expected to provide (and fund) the long-term efforts to maintain and evolve products; this is best handled within the commercial sector. Given that the STARS Ada/X

[1]™ X Window System is a trademark of the MIT X Consortium.

bindings will also be eventually transitioned to commercial support, this report provides some guidance as to what programmers and customers can expect from eventual commercial Ada/X products. Laying the foundation for commercialization also means preparing for formal and *de facto* standardization efforts as system developers and customers work to assure portability of applications and programmers across different implementations. This report discusses what various standards boards might do once the STARS Ada/X work and other Ada/X efforts move towards commercial acceptance.

In addition to the software development community that must program with Ada/X, there is also the system acquisition community that is responsible for specifying these applications and reviewing their products. This paper provides an overview of the X Windowing System in general, so that the parts of the STARS Ada/X binding can be seen in this larger context. This paper will also review some of the standard application development in X roles that apply to both Ada and C development with X. Knowing how these roles work provides an acquisition person with an idea of what to look for from particular application development personnel (in terms of training, experience, and resources) in order to reduce development risk.

ACKNOWLEDGMENTS

This document has been prepared by The MITRE Corporation under Project No. 5800, Contract No. F19628-89-C-0001. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, United States Air Force Hanscom Air Force Base, Massachusetts 01731-5000.

The author wishes to thank Marlene Hazle, David Emery, and Richard Hilliard for their support during this work and for their helpful reviews of early draft documents. The author also wishes to thank the following members of the contractor development teams for their information and advice during all our meetings: Mark Nelson and Howard Turner (SAIC), Robert Smith and Timothy Schreyer (Unisys), and David Jones, David Wilson, William Halley, and Robert Rosen (all of Boeing). Finally, an acknowledgement to Linda Gaudet and Joan Lavery for all their work in editing and preparing this document.

TABLE OF CONTENTS

SECTION		PAGE
1	Introduction	1
2	Historical Background	5
2.1	Overview of X Window System	5
2.2	Earlier Ada/X Bindings	7
2.3	Initial STARS Ada/X Bindings	7
2.4	Other Ada/X Development Efforts	9
3	Ada/Xlib	11
4	Ada/Xt	15
4.1	Xt Callbacks	15
4.2	Widget Packaging	21
4.3	Resource Manager	23
4.4	Widget Typing and Subclassing	28
5	Future Development Work	31
6	Potential Standardization	33
7	Distribution Issues	37
8	Acquisition Guidance	39
List of References		41
Appendix A	Detailed Ada/Xlib Changes	43
Appendix B	Detailed Ada/Xt Change	55
Glossary		87
Index		89

LIST OF FIGURES

FIGURE		PAGE
2-1	X Window System Stack Architecture	5
2-2	X Client/Server Architecture	6
3-1	Ada/X Source Directory Structure	12
4-1	Widget Task Instance	16
4-2	Internals of Widget Task	17
4-3	Propagation of Calls to Parent Widget Task	17
4-4	Propagation of Calls from Parent Widget Task	18
4-5	Display of Widgets on Workstation Screen	18
4-6	Combination of Widget and Application Design Units	19
4-7	Relationship between a Widget's Packages	21
4-8	Relationships between Widget Hierarchy and Application	22
4-9	Simplified Relationships between a Widget's Parts	23
4-10	Relationship of Intrinsics to Widget Hierarchy	24
4-11	Widget and Resource ADT Relationships	25
A-1	Problems with Mask Data Types	44
A-2	Examples of Constant Object Definitions	47
A-3	Example of String Conversion Function	50
B-1	Ada/Xt Subprogram Pointer Template	55

LIST OF TABLES

TABLE		PAGE
1-1	List of TIM Dates and Locations	1
1-2	Ada/X Binding Development Roles	2
2-1	Ada/X Binding Efforts	8
B-1	Widget Record Layouts	61
B-2	Allocation of Subprograms to Packages	66

SECTION 1

INTRODUCTION

The STARS program is interested in the commercial acceptance of the products its contractors develop. With the growing popularity of the X Window System as a complement of many application systems, the STARS program has been interested in bindings between Ada and the X Window System for a number of years. The early STARS Foundations work done by SAIC on an Ada/X binding proved to be a seminal project to many future Ada/X prototypes and applications. As X Window System technology advanced, the STARS program wanted to update a binding based on lessons learned from the STARS Foundations binding and new functionality needs.

Ada programmers wanted to minimize the learning curve when applying the X Window System to their applications. The diverging collection of Ada/X bindings, some derived from the original STARS Foundations work, was proving to be a barrier to portability and reusability of applications. The STARS program decided that those Ada/X binding efforts being done by its contractors and subcontractors should be convergent, so a task was started to develop a common STARS Ada/X binding applicable to general Ada software development. This task would stress cooperative development and high levels of technical interchanges.

Four TIMs were held in 1990 to bring together representatives from the STARS contractors (both prime and subcontractors) and The MITRE Corporation to work out Ada (DOD 83) and X technical issues that were too complex or controversial for electronic mail. The dates and locations for these TIMs are shown in table 1-1:

Table 1-1. List of TIM Dates and Locations

DATE	LOCATION
23-25 July 1990	Unisys, Paoli, PA
8-10 August 1990	SAIC, San Diego, CA
19-21 September 1990	Unisys, Paoli, PA
15-17 October 1990	Boeing, Renton, WA

The first of these TIMs started with the two incompatible (SAIC and Unisys) Ada/X bindings and began working towards first identifying and the resolving the differences between the two implementations. After each of the four TIMs, a detailed report of the changes made to these initial Ada/X bindings and the progress towards the common specifications was prepared. This report was extracted from the four earlier TIM reports.

This report presents the STARS Ada/X binding at two levels of detail. The main body of this report contains a high-level overview of the changes and implementation choices. A detailed list of all the changes is available in the two appendices to this report.

This report should be useful in a variety of software development roles that may have to use or interface to an Ada/X binding. Table 1-2 below lists some of these roles. The binding developers have the toughest role because they must understand all the implementation dependencies of their target Ada compiler(s) as well as the intricacies of X's OOD and OOP approaches. Binding developers should have very high programming skill levels; they will be the most interested in the detailed implementation choices in the appendices. Future binding developers will include commercial Ada compiler vendors who wish to provide Ada/X as a Commercial Off-The-Shelf (COTS) product, particularly since the vendor will best know how to exploit the OOP support provided by their Ada compiler.

Table 1-2. Ada/X Binding Development Roles

ROLE	DESCRIPTION
Binding developers	Ada programmers who create Ada/X bindings, such as this one,
UIMS developers	tool developers who need to interface to these bindings,
Widget developers	programmers creating reusable pieces of (Ada) window code,
Application developers	programmers using the above products in their domains, and
Acquirers	those responsible for checking all these earlier products.

The developers of User Interface Management Systems (UIMS) are another small but select group of programmers who will need to know the details of Ada/X. A trend among X application developers is to use UIMSs as a front-end Computer-Aided Software Engineering (CASE) tool so the applications programmers do not have to learn the intricacies of X programming through subroutine calls. Instead, a graphical tool is used to lay out the position and appearance of display screen entities, and the necessary subroutine calls are generated automatically. Several Ada UIMSs are under development, such as the National Aeronautics and Space Administration's (NASA) Transportable Application Environment Plus (TAE+) (Szczur, 1990), and the Software Engineering Institute's (SEI) Serpent (SEI, 1989) system. Developers of these Ada UIMSs will need to know the implications of using different Ada/X subprograms and types in their generated code.

One goal of OOP is to produce reusable pieces of code that future programmers will find useful. With the X Window System, these reusable code pieces that encapsulate the visual appearance, end user interaction, and Applications Programming Interface (API) are called widgets. Because X widgets can be used to provide a common look and feel to applications

and because they can be used to represent (to both the end user and the application's programmer) the concepts of an application domain, these widgets are often written by widget developers who are experienced in a given domain. When using Ada/X, these widget developers will have to understand how the OOP concepts of X have been implemented in Ada to create or reuse widgets that are consistent with other Ada/X widgets.

The application developers (and eventually system acquisition people) will have to write their Ada programs on top of whatever capabilities the binding developers and widget developers provide. An Ada/X binding should limit the implementation complexities that the application developers must deal with; the programmers should be presented with clean interfaces by the binding developers and widget developers so the common OOD and OOP paradigms the application developers learned about X will still apply. Application developers will the system acquisition people who review their work will have to know the overall concepts of Ada/X, but should not have to know all the implementation details.

As a guide for the readers of this report, they should first review table 1-2 and decide which of those roles best applies to them. The four developer roles will be interested in the technical details of appendix B, particularly given the importance of widgets. Developers who will be working with non-widget X interfaces will be interested in appendix A.

Any reader who has not had much exposure to X in general and Ada/X in particular, should read the historical background (section 2). The acquirers will be particularly interested in the sections on future development work, distribution issues, and conclusions (sections 5-8). Any of the development roles will be interested in all of these sections.

SECTION 2

HISTORICAL BACKGROUND

2.1 OVERVIEW OF X WINDOW SYSTEM

There are several different ways of viewing the X Window System. Figure 2-1 shows the overall X system architecture arranged in a stack hierarchy.

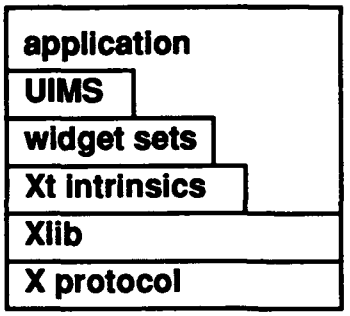

Figure 2-1. X Window System Stack Architecture

The lowest level of this stack architecture is the X protocol, which defines the flow and structure of packets of information around a network. A library of subroutines (known as "Xlib") is used to create and receive these packets. These Xlib subroutines provide all the common actions associated with a bit-mapped display terminal, such as drawing a line or being notified that a keystroke was depressed.

Application developers soon realized that trying to write a complex application through these low-level Xlib calls was an extremely complex task. Eventually, a layer on top of Xlib was built, known as the intrinsics level. The intrinsics were written to provide an OOP interface. A major change in an intrinsics layer is the use of event-driven callback routines. No longer does the application developer call the windowing system from a main line programming loop provided by the application. Instead, the intrinsics layer contains an application's main line routine; the application developer provides the address of subroutines to call back in response to certain events. This completely inverts the flow of control and data in an application, where OOP message passing becomes the main programming paradigm. The most popular set of intrinsics functions, used on almost all X applications, are the X toolkit ("Xt") intrinsics provided by the Athena consortium.

The Xt intrinsics provide just a set of OOP routines and conventions for defining actions, such as how workstation events are passed to the application and how changes are propagated through the callback functions. The intrinsics' objects (where an object is a data store and a

related collection of manipulation subprograms) are known as widgets in X's terminology. To ensure a common look and feel, as well as provide application developers with a large set of reusable components to start with, several organizations have written widget sets. Some widget sets, such as the Athena widgets, are provided at no extra charge on the X distribution tapes. Other widget sets are developed and maintained for sale by commercial organizations. The two best known of these X widget sets are the Open Software Foundation's (OSF) *Motif*™[2] widgets and UNIX®[3] International's (UI) *Open Look*™[4] (OL).

As discussed earlier, some application developers have found using widget sets such as Motif and OL to be still too low-level on which to develop programs. They have chosen the option of using a UIMS to generate the API calls to a widget set. The highest level of an X architecture remains the application code, which has the option of using one or more of the lower layers. These lower layers are typically provided to the application as program libraries that are connected at link time, based on which API calls were used in the application.

Another way of viewing X's architecture is through a client/server model. Figure 2-2 below shows a simple example of a client/server architecture.

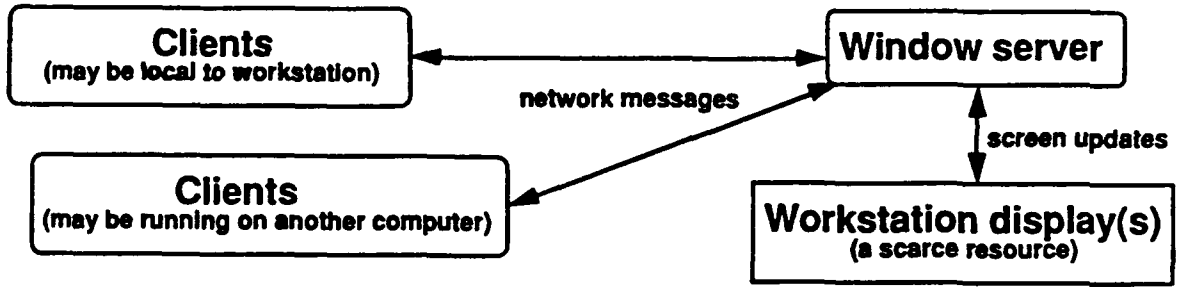

Figure 2-2. X Client/Server Architecture

The example above shows how an X server is responsible for managing a scarce resource, in this case the workstation's graphical display surface and input devices (keyboard and mouse). The application programs (shown at the top of figure 2-1) are the clients in figure 2-2, communicating with the server through X protocol calls. These clients can be local (running on the same workstation the server is) or can be running on another computer (communicating over a network).

2™ OSF and Motif are trademarks of the Open Software Foundation.

3® UNIX is a registered trademark of AT&T Bell Laboratories.

4™ Open Look is a trademark of UNIX International.

The X architectures shown above provide a framework or reference model for describing X software. The actual X Window System software that is sold or distributed can change as new features are added and bugs are fixed. Over time, the Athena Consortium (the parent organization for the X Consortium) has made both major new versions of X and minor new releases of individual versions available for distribution. At the time of this report's preparation, the current major version of X was #11 and the release level was #4. This is commonly abbreviated as X11R4. Note that even when the Athena Consortium releases a new version of the base (or reference) X server implementation and client libraries (such as X11R5), there may be a lag time before a commercial widget developer (such as OSF or UI) or a language binding developer (such as for Ada/X) will upgrade their products to conform to this new release. Therefore, understanding how version skew can affect these X layers and the programming language bindings to them is important.

2.2 EARLIER ADA/X BINDINGS

There have been a variety of efforts to create Ada/X bindings; table 2-1 lists just some of those the author has heard about.

In the table below, the Binding column indicates to which level of the X stack architecture (Xlib, Xt, and/or widget set) the interface is being bound. The 'R' number indicates for which X release level the Ada API is being provided, so 'R3' indicates X11R3. The Developer column indicates who is (or has been) developing this Ada/X binding and what year that binding was released for use. As can be seen in table 2-1, most of the Ada/X binding development work has been done in the last few years.

2.3 INITIAL STARS ADA/X BINDINGS

The STARS program has been involved with the development of Ada/X bindings for a number of years. The earliest work was done under what is now known as the STARS Foundations work. SAIC was funded to develop an Ada/Xlib binding to X11R2 as well as an implementation of HP's Xray widget set. As with the other STARS Foundation's products, these were generally made available to the contractor and eventually the whole Ada community. One commercial vendor, GHG, took this X11R2 binding and packaged it into a product. This STARS Foundation's version of Ada/X has been the basis of much of the Ada/X prototyping and development work that has been done so far.

The most recent phase of the STARS program included work to extend the initial (X11R2) STARS Foundation's Ada/X binding to up-to-date releases of X, and to provide an implementation of a widget intrinsics level that is more generally accepted by programmers than Xray. SAIC (as a subcontractor to Boeing) and Unisys were funded to implement this improved version of Ada/X.

Table 2-1. Ada/X Binding Efforts

BINDING	DEVELOPER
Ada/Xlib (R2)	SAIC for original STARS Foundation work (1988),
Ada/Windows	DEC's interface to their Xlib-like DECWindows™[5] product (1989),
Ada/Xlib (R3)	GHG's (and others) commercial version (1989),
Ada/Xlib (R3)	current version released by SAIC (1990),
Ada/Xlib (R3)	current version released by Unisys (1990),
Ada/Xlib (R4)	merged version of SAIC and Unisys work (1991),
Ada/Xt (R2)	Software Productivity Consortium (SPC) proprietary binding to Hewlett-Packard (HP) widget set (1989),
Ada/Xt(R3)	initial Xt specifications and bodies from SAIC (1990),
Ada/Xt (R3)	initial Xt specifications and bodies from Unisys (1990),
Ada/Xt (R3)	forthcoming merged versions with common specs. and different bodies [both SAIC and Unisys] (1991),
Ada/Xt (R4)	Jet Propulsion Laboratory (JPL) work to create Xt binding (1991),
Ada/Motif (R3)	OSF-funded Motif bindings work at University of Lowell (1991),
Ada/Motif (R3)	NASA's TAE+ UIMS uses Ada bindings to Motif (1991),
Ada/Xt (R4)	SEI's Serpent UIMS, generating an Ada interface (1991),
Ada/Xt (R3)	TRW's Generated Reusable Ada Man-Machine Interface (GRAMMI) UIMS (1991), and
Ada/Motif (R3)	TeleSoft's TeleUSE UIMS uses Ada bindings to Motif (1991).

Initially, SAIC and Unisys were working independently towards individual (and incompatible) versions of Ada bindings to XlibAda implementations of Xt and widget sets that sit above Xt. With the change in direction in STARS management and the increased emphasis on standardization and commercial acceptance of STARS products, it was decided that developing incompatible Ada/X bindings was not a very good idea. By this time, SAIC and Unisys had completed their designs on their respective versions of Ada/X and had nearly completed the coding.

A new STARS task was initiated to merge the interfaces between these two implementations. The two completed but incompatible implementations were still released in 1990, but mainly to serve as prototypes and proof of concept for the final STARS Ada/X product to be released in 1991. Meanwhile, SAIC, Unisys, Boeing, and MITRE representatives would hold a series of TIMs to develop common specifications towards a single Ada/X interface. This report documents the progress of those meetings.

---

[5] ™ DECWindows is a trademark of Digital Equipment Corporation.

2.4 OTHER ADA/X DEVELOPMENT EFFORTS

Many of the Ada/X developments discussed above, including STARS', are based on the idea of implementing all or parts of Ada/X as a binding on top of an underlying C/X implementation. Not all Ada developers agree with this assumption; they believe that it is possible to develop an all-Ada implementation of the entire X Window System so the Ada programmer is not tied to what the base C/X implementation provides, and Ada's built-in OOD and OOP constructs can be fully exploited.

Rational Corporation has developed an (almost) all-Ada implementation of Xlib for both their R1000™[6] computers and Sun 3™[7] workstations. This is instead of the more traditional approach of writing a binding to an underlying C/X implementation. They do have some assembler at the lowest layers to interface to protocol, where direct calls to network sockets are made. While Rational does use the same Ada/Xlib package specifications as the joint STARS Ada/Xlib product, Rational believes it is possible to implement the STARS Ada/Xlib binding through calls to the Rational Ada/Xlib implementation. Rational plans to turn this ~33,000 line Ada/Xlib (X11R4) implementation into a supported product, and donate the source code the Athena Consortium for distribution on the X11R5 release tape.

All the Ada/X development work discussed above concentrates on the client side of X's architecture (see figure 2-2), providing the Ada interfaces and libraries that an application needs. But there has been work at Sanders/Lockheed to develop an all-Ada implementation of an X server. See (Lewin, 1989) for a description of how this Ada/X server implementation was done.

---

[6]™ R1000 is a trademark of Rational.

[7]™ Sun 3 is a trademark of Sun Microsystems.

SECTION 3

ADA/XLIB

This section introduces the STARS Ada/Xlib binding to potential users. As figure 2-1 indicates, there is less application development work in this area but Xlib forms the foundation on which the higher layers are built.

The STARS developers had decided that developing and testing the Ada/X code using the Ada compilers of Verdix Ada Development System™[8] (VADS), Digital Equipment Corporation (DEC) DECAda™[9], Alsys, and TeleSoft on various UNIX and VAX/VMS™ workstations would cover much of the currently installed Ada developer base. This also corresponded to the Ada compiler and host platforms that the STARS developers had ready access to. But there was the recognition that there were other Ada compilers that this Ada/X code has not been tested against. While the STARS developers believe that an Ada/X implementation that could work on this representative set of compilers would also work on other compilers, any future commercialization and standardization work may include working out any (unintended) compiler dependencies left in the STARS code.

One issue discussed during the TIMs was the best way of packaging the Ada/Xlib (and eventually Ada/Xt) packages into Ada source files, and how to distribute those source files among a host's directory structure. One alternative would be to use a single directory tree structure with the different Ada compiler vendor-specific packages stored in files with unique names. For example, the `X_Lib_Interface` package's specification is stored in the files `x_int_alsys_.a`, `x_int_tele_.a`, `x_int_vads_.a`, and `x_int_vax_.a`, one for each reference compiler vendor (Alsys, TeleSoft, Verdix, and DECAda). A few target compilers require their own files, as with `x_int_vads_mips_.a`. There are only three source files that have to be separated this way, so initially this would appear easy. But note that this places duplicate copies of the same Ada package (`X_Lib_Interface` in the example above) in the same directory; this can confuse some Ada compiler automatic recompilation systems.

The alternative the STARS developers chose was to isolate these Ada source files in their own (sub)directories. Since there might be further specialization necessary for these files (as with new "root" Ada compilers, major version upgrades among existing ones, and/or unusual X target workstations), a directory structure was deemed easier to expand than a complex name encoding scheme for the source filenames. Note that this scheme will not work if an uninformed Configuration Management (CM) organization decides to squeeze all the files into a common directory (which happened in an earlier STARS Ada/X release, and

---

[8]™ VADS is a trademark of the Verdix Corporation.

[9]™ VAX/VMS and DECAda are trademarks of Digitial Equipmant Corporation.

resulted in some source files being overwritten). The delivered source files will be organized on the tapes, for example in a directory structure that looks something like what is shown in figure 3-1.

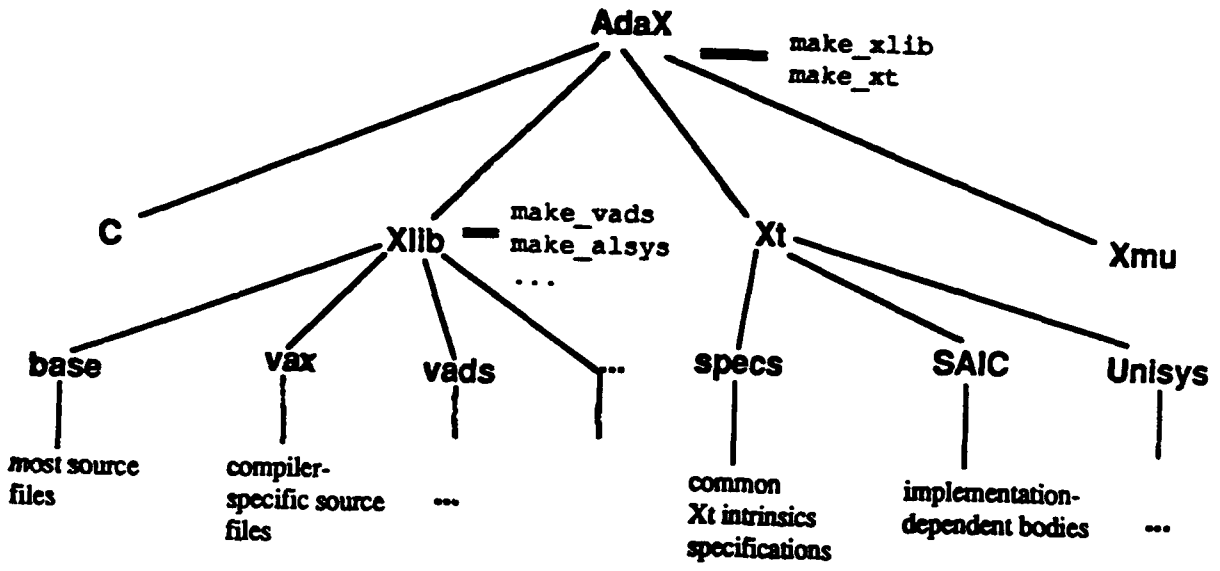

Figure 3-1. Ada/X Source Directory Structure

Note that the script files necessary to build a particular X Window System programming level (such as `make_xlib` and `make_xt`) and a particular compiler's version (such as `make_vads` and `make_alsys`), are placed one or more directory levels from the source code. These script files are intended to capture only the most important information, such as the compilation order. Users of Ada/X would have to supply their own creative solutions for versioning, conditional compilation, etc. Note that Xmu holds utility functions.

There were several design goals the developers of Ada/Xlib wanted to provide or avoid. One Ada programming construct that is commonly used in other Ada/X bindings is `system.address`, as the data type of many parameters and functions used in the underlying C/Xlib implementation. The STARS binding developers felt that this was a non-portable construct that would become a major maintenance problem. Few if any uses of `system.address` appear in any of the Ada/Xlib package specifications.

One decision made by the STARS binding developers was to develop a merged and common implementation (the Ada specifications and bodies) of the binding to the Xlib layer of X11R4. Working on a common SAIC and Unisys Ada/Xlib allows the binding developers

to create these bodies only once. The same Ada/Xlib code – consisting mainly of Ada specifications and `pragma Interface` calls to underlying C/Xlib implementation – would be available on both companies' distribution tapes. Working on a common Ada/Xlib also means that STARS can release a complete "product quality" system (the completed Xlib specifications and bodies) for potential customers and potential standardization.

The four TIMs covered many detailed technical issues and changes that had to be made in Ada/Xlib. (See appendix A of this report for a description of each of those changes.) While these historical changes may be of limited interest to programmers who have not had any exposure to earlier Ada/X products (such as those directly or indirectly derived from the STARS Foundation's work), binding developers and widget developers may be interested in some of the pitfalls to avoid and the rationales of why STARS Ada/X was built the way it is.

SECTION 4

ADA/XT

This section provides an overview of different Xt features and how the two STARS Ada/Xt implementations chose to provide them. Each of these features is crucial to Xt applications, so any reader should study them carefully.

Because SAIC and Unisys had developed (coded) much of their respective implementations before this Ada/X integration task began, there was a limit imposed by schedule and funding as to how much of the existing code could be modified. Both SAIC and Unisys had decided to create an Ada/Xt implementation (the specifications and bodies) instead of just using a binding to C/Xt (such as what Ada/Xlib used with C/Xlib) because of portability issues and to exploit Ada's OOD/OOP features. As a result, there was a large amount of Ada/Xt code that could undergo drastic revisions in the time available. For example, the premerged Ada/Xt code was written to the X11R3 intrinsics. At the time the TIMs began, the Athena Consortium had released a major revision to the intrinsics in X11R4. The STARS binding developers decided to keep the Ada/Xt intrinsics subprograms at X11R3 instead of upgrading to X11R4 (as was done with Ada/Xlib) because of the scope of the changes needed. Later in this report there will be a discussion of the impact of this decision, especially in dealing with standardization and widget developer work.

Appendix B of this report contains a discussion of the major technical changes made in Ada/Xt in order to provide a common specification. SAIC and Unisys had elected different Ada design and coding approaches for implementing Xt. This section of the report will cover some of the higher level Xt and widget design concepts necessary before any detailed Ada/Xt discussions. All the Ada/X roles listed in table 1-2 will need to understand these issues. Binding developers may be the major audience interested in appendix B.

4.1 XT CALLBACKS

The technical difference between SAIC and Unisys implementations is how Ada/Xt callbacks are implemented. Xt intrinsics use an event-driven model of execution; the widget contains a pointer to an application routine that is called when a specified event occurs. The main control of the program exists within the internals of Xt; almost all of the application's code will be subroutines that are called from within Xt. Unfortunately, Ada doesn't handle this particular OOD concept of being able to point to instances of encapsulated code and data very well, so both Unisys and SAIC had to design something that would simulate this concept.

SAIC's approach is to use tasks and task types to encapsulate widget instances. Since Ada allows pointers (access) to task types, it is possible to set up and assign pointers to which tasks to call when certain events occur. Ada task types can be dynamically created and terminated, so the application developer has some flexibility on how and when to construct widgets.

Since Ada tasks exist within Ada's strong typing model, that means there are normal restrictions on how data is passed as parameters to the task's entry points and even how tasks of different types are treated separately. This is a problem with Xt intrinsics because a weak typing model is assumed. Applications programmers and widget developers are supposed to be able to easily construct higher level widgets from the pieces of lower level widgets. Xt's "inheritance hierarchy" assures that when an event occurs within one widget, it is propagated up the hierarchy to all the other related widgets that also need to deal with that widget. Widget developers provide default methods and "values" that the application programmers can use to control what is done when these events occur. Application developers are free to override these default methods through subclassing of a widget class into a new subclass and/or by manually setting a new method that is run as an alternative to the old one.

All this dynamic definition and reconfiguration of the structure or architecture of who is allowed to call what tends to conflict with Ada's strong (task) typing model. SAIC's solution is to separate the different types of widgets and their associated (underlying) Xt intrinsics into different Ada task types. But instead of different entry points for the different methods of this type of task (widget class), there's only one entry named `handle` for all tasks (with each defined `handle` having exactly the same parameter profile). The advantage of this approach is that Ada's `Unchecked_Conversion` procedure can (hopefully) be used to convert a pointer of one instance of these task types to another. By converting a task access from one type to another, the same calls to entry `handle` can be used to simulate the overwriting or subclassing of one Ada/Xt "method" by another. Figure 4-1 shows what such a task would look like using the Buhr (Buhr, 84) notation.

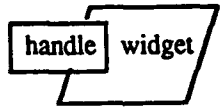

Figure 4-1. Widget Task Instance

Looking at figure 4-2, we see the internal details of the widget where a call to entry `handle` results in a call to one of the method handlers that have been defined within the widget task.

The invocation of one of these methods results in an Ada subprogram being run, or it might result in a call to the task that manages the widget's class being called so a predefined class method is invoked. In figure 4-3, we see that method #2 will result in a call to a subprogram, while other methods are passed up the inheritance chain (indicated by the thicker arrows) to the parent or superclass widget class task.

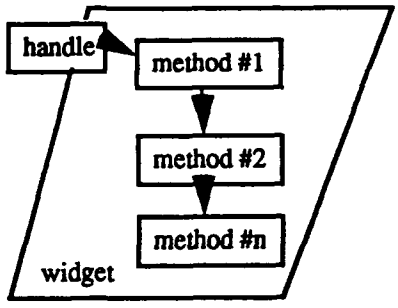

Figure 4-2. Internals of Widget Task

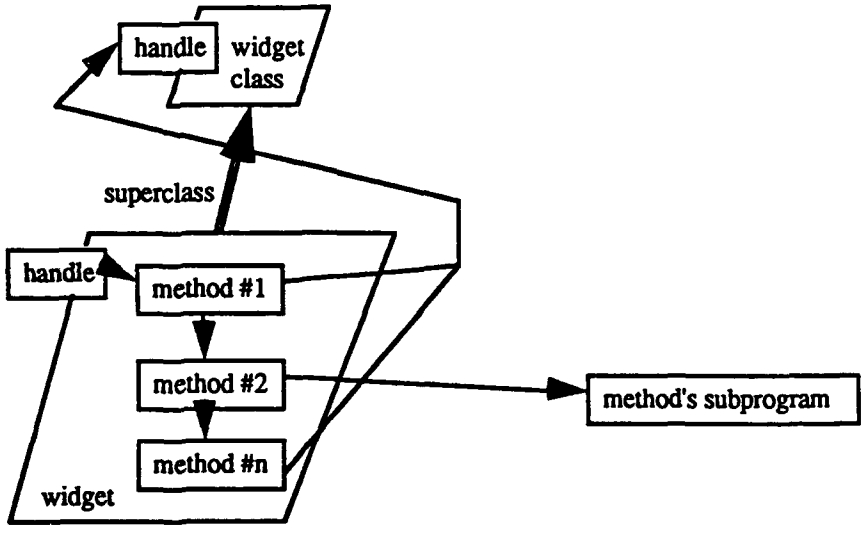

Figure 4-3. Propagation of Calls to Parent Widget Task

In addition, a widget class (superclass) may have some of its methods require that information and calls be propagated down to all the children ("instances") of this class. In figure 4-4, we see that three different widgets (numbered #1 through #3) have been created earlier from this class. It turns out that method #1 in the widget class has to call all the children of this class, since they need to be informed of some activity as well. So figure 4-4 shows the superclass task calling each of the children tasks.

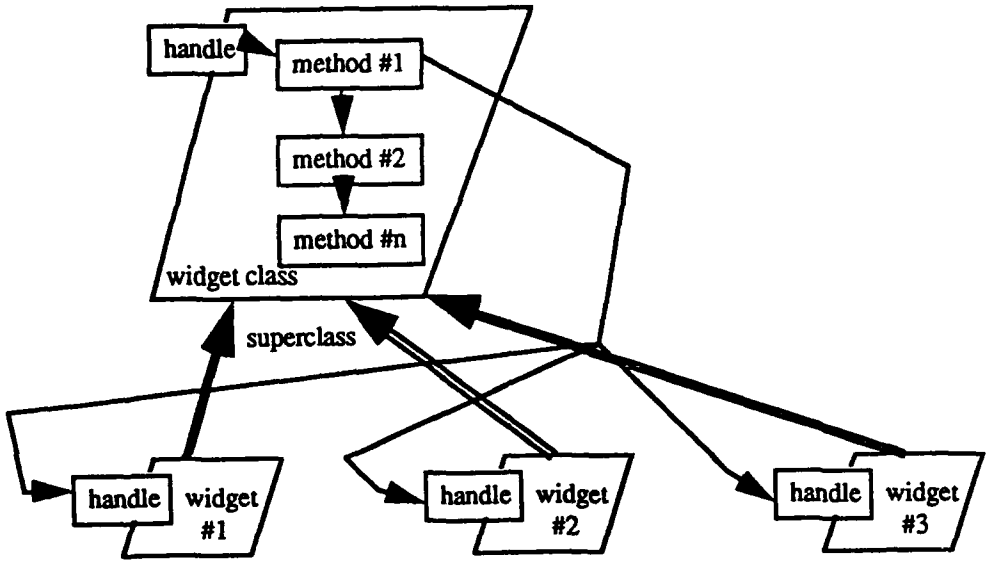

Figure 4-4. Propagation of Calls from Parent Widget Task

Figures 4-4 through 4-11 are simple examples of a widget class hierarchy from a widget developer's point of view. But the application developers will have their own points of view, based on the actual usage of widgets within an Ada/X program. For example, let us assume the widget class described in figure 4-4 was for some sort of shell widget that happens to have an enclosing box or outline around it. The different styles of enclosing boxes available are represented by widgets #1 - #3 in figure 4-4. Figure 4-5 shows the window or screen the application programmer wants to create.

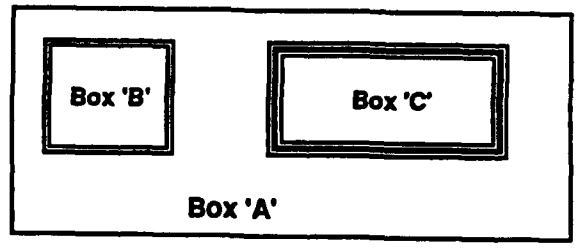

Figure 4-5. Display of Widgets on Workstation Screen

Box 'A' is an instance of widget #1 (from figure 4-4), box 'B' is an instance of widget #2, and box 'C' is an instance of widget #3. The application work to be done whenever the workstation's cursor is in any of these boxes is allocated to a particular application program

task (that might be doing something else when not handling window events). From the application developer's point of view, the design of this application would look like figure 4-6.

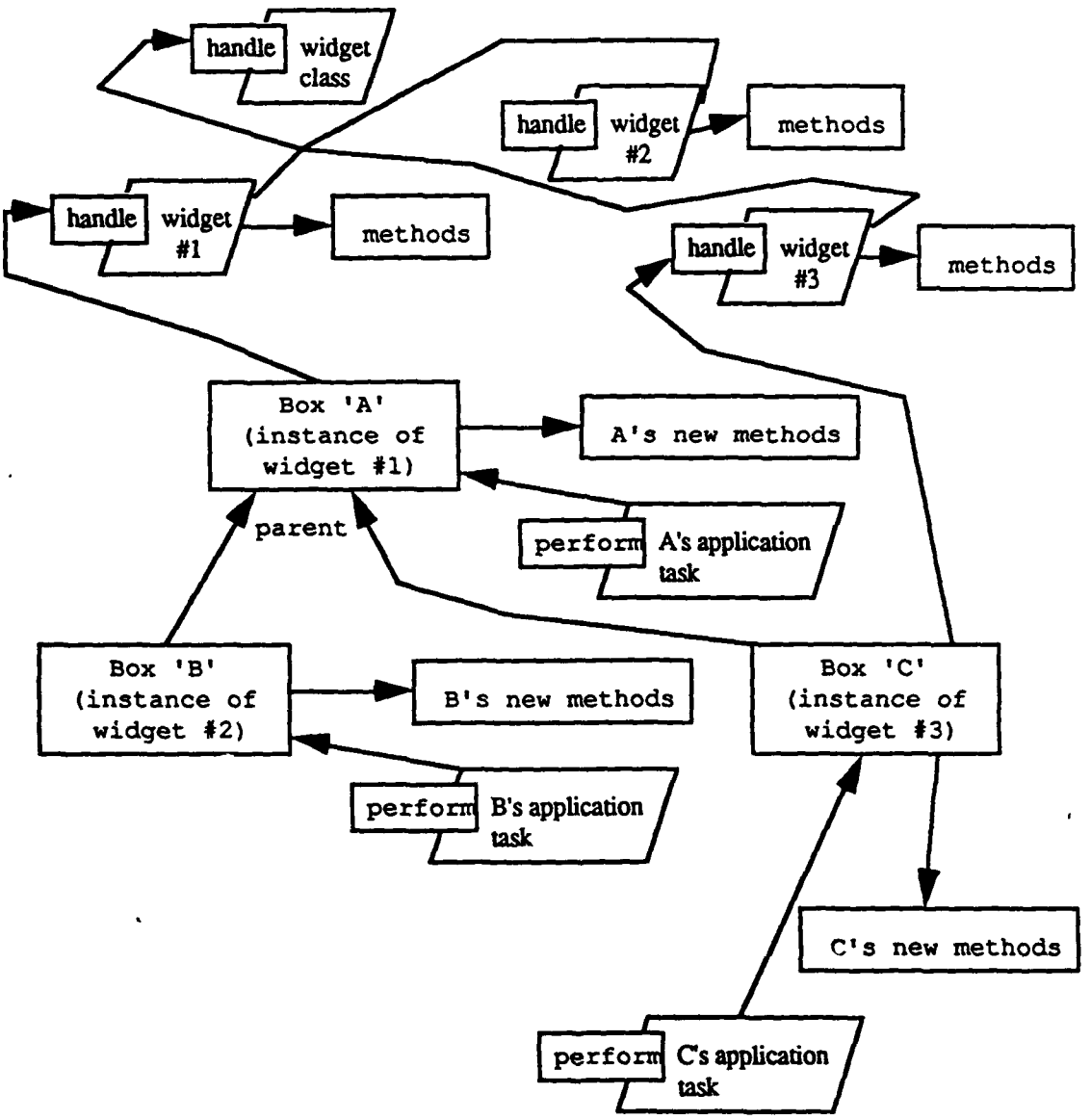

Figure 4-6. Combination of Widget and Application Design Units

The class hierarchy originally defined by figure 4-4 has been expanded to include application widget instances. As in figure 4-3, figure 4-6 shows that either the widget subclasses

(widgets #1 - #3) or the application's widget instances (Boxes A - C) could override one or more of the methods (or actions) defined in the original widget class. The various options of Xt event propagation, method subclassing, predefined versus user-defined methods, and dynamic widget instance creation will lead to complex application design diagrams. For example, the thread of behavior that results from a single event (such as a mouse click) might be propagated through Ada/Xt default methods, widget class methods, (new) widget instance methods, and applications code in a variety of orders. As Ada/X application developers attempt to use this combination of Ada units (particularly when debugging the software), they will need some way of organizing their design choices.

The diagrams above have shown only a very simplistic overview of the tasking structure. In reality, there are multiple layers to these diagrams, so the hierarchies and the connection arrows would be much more complex. In addition, the packages, subprograms, and tasks that the application developer creates would also become part of an overall (graphical) diagram. Widget developers might have to see this entire structure, while end-application programmers might only have to see the bottom layers that provide the most useful widget (tasks).

The architecture from both the applications programmer's point of view and the widget developers point of view can be very dynamic, with the links between Ada tasks (widget classes and instances) subject to adjustment. From MITRE's experience developing the Descriptive Intermediate Attributed Notation for Ada (Evans, 83) (DIANA) Query Language (DQL), which makes extensive use of dynamically allocated tasks arranged in hierarchies, the authors have learned how hard it can be to conceptualize the design of how all these tasks fit together (Byrnes, 1988, 1989). SAIC intends to document the interactions between the predefined Xt (and widget) tasks and task types, and the application's tasks in their forthcoming *Ada/Xt User's Guide*, but it will have to be a very detailed document to explain everything to Ada widget developers and applications programmers.

A similar design issue exists for the Unisys Ada/Xt callback scheme. Instead of using access to task types, they use pointers to Ada procedures. Since there is no such thing as an access pointer to an Ada subprogram, Unisys uses the `ADDRESS` attribute of the procedures as the values to place and adjust within widget methods. Unisys also uses a generic callback package, where the action routines are passed into it as a generic procedure parameter. The work involved to overwrite default methods will require passing `system.addresses` around as further parameters. As anyone who has programmed in C/Xt can tell you, a variety of terrible things will happen if these pointers become corrupted or set incorrectly. (See the Unisys document (Wallnau, 1990) for the diagrams that define the relationships between these packages.)

Unisys tries to bring some order to the confusion caused when one start passing `subprogram'ADDRESSes` around. There is a single dispatch subprogram that is passed a private data type which deals with the methods, that centralizes where these callbacks are handled. But there could be confusion in the minds of application developers about how Xt events are propagated through the widget classes and instances because there are no tasks to isolate all these threads of control.

4.2 WIDGET PACKAGING

As discussed earlier, an important part of X in general, and Xt in particular, is the support for encapsulated code collections known as widgets. While many widgets are provided commercially by widget developers (such as those for Motif and OL), some of the base widgets that all these other widgets are built in (inherit from) are considered part of Xt. This section describes how Ada/Xt implements these common widgets.

Ada/Xt had to deal with the interactions between the intrinsics packages and the Ada packages that contain the definitions of the standard or base widgets that are part of Ada/Xt. The `XtCore` widget is one whose behavior is heavily involved with the intrinsics packages. The other widgets (such as `composite`, `constraint`, `shell`, etc.) are built on top of `core`. Each of these widgets is in its own Ada package, where the STARS binding developers decided to follow Unisys' approach of using "private" packages to encapsulate the widget developer's information. So for a given widget, the Ada source code for it will be in the source files `widget_.a` (the package specifications intended for use by everyone), `widget_p_.a` (the [semi]private package specifications containing the [overloaded] definitions, layouts, and conversion functions used only by the widget developers), and `widget.a` (the package bodies that implement the specifications).

The figures below show how these three Ada packages are connected to one another, to other widgets in the hierarchy (which will eventually involve inheritance), and to the eventual applications code. Figure 4-7 shows a simple Buhr-style diagram, with the arrows representing the `with` relationships between units.

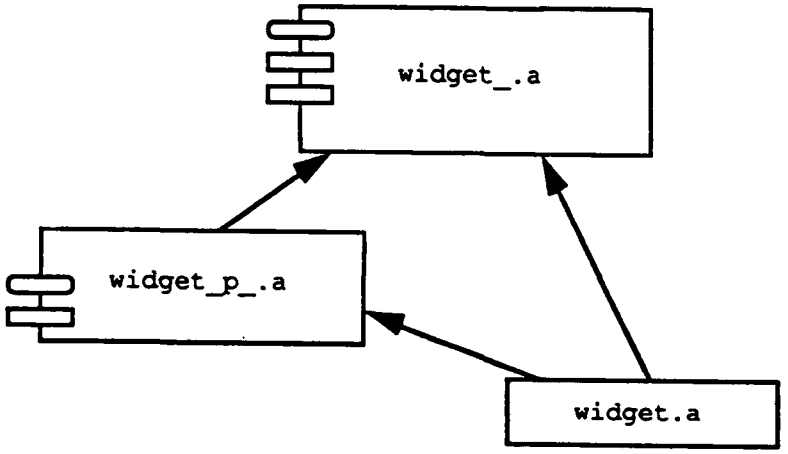

Figure 4-7. Relationship between a Widget's Packages

Xt widgets are built on top of each other in a hierarchy, so another widget (named `widget2`) would be built on top of this first widget. Figure 4-8 shows the relationships between the public and semi-private parts of this part of the widget hierarchy. Note the `with` relationships

between both the public package specifications (`widget2_.a` → `widget_.a`) and the semi-private package specifications (`widget2_p_.a` → `widget_p_.a`). Also note the unit of applications code that `withs` in `widget2` (through the arrow to `widget2`). As far as the application developer is concerned, the connections to `widget2`'s parent (`widget`) are implicit and hidden. This means that the widget developers have to be careful with the `pragma Elaborate` statements as this widget hierarchy grows (to be sure the intrinsics body is elaborated) so there are no unpleasant surprises for the applications programmers.

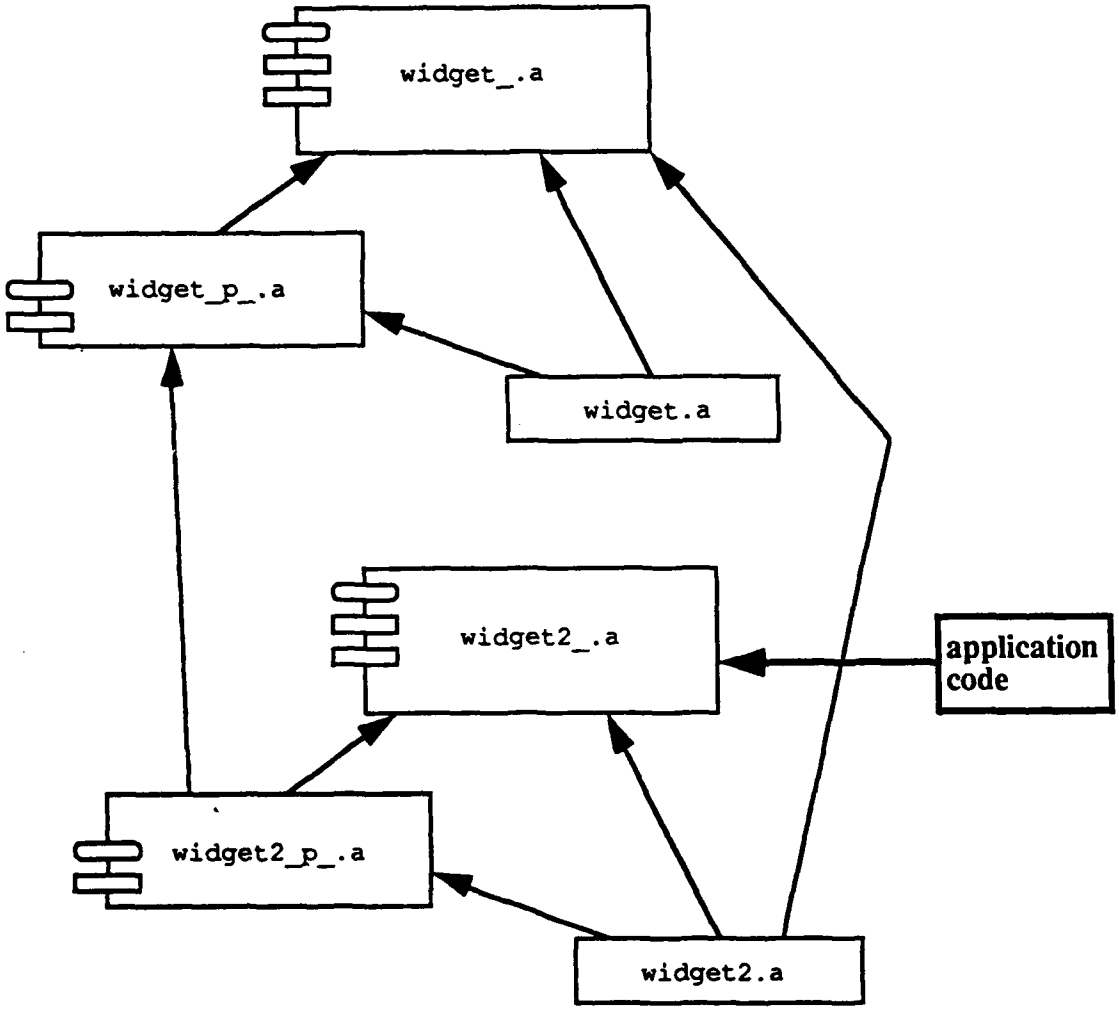

Figure 4-8. Relationships between Widget Hierarchy and Application

As the widget hierarchy becomes more deeply nested, one application design-level question is how to abstractly represent some of this information without overwhelming the

programmers with detail. One idea is to contract the three parts of a widget's source code (`widget_public_`, `widget_private_`, and `widget_public`) into a small "triangle" that simplifies the `with` relationship diagram shown above in figure 4-7. Figure 4-9 shows such a diagram for one widget.

Figure 4-9. Simplified Relationships between a Widget's Parts

The `core` widget is generally considered the base or top-level widget in the hierarchy of classes, although there are four lower level classes (the `Object`, `Rect_Object`, `Window_Object`, `Composite_Object` and classes) that provide the abstract foundations used for non-widget entities in recent versions of Xt (the light weight gadgets). So the `core` widget is built on top of these two classes and all other widget (classes) are then built on top of `core`. The Ada/Xt `Xt_Intrinsics` package is connected to this hierarchy through `with` relationships established with the `core` widget. Figure 4-10 uses the simplified diagram introduced in figure 4-9 to show this hierarchy.

4.3 RESOURCE MANAGER

X resources are an attempt to provide strongly typed data that follows the class and inheritance hierarchies of the widgets, allowing the X user to select (through conventional editing of their `~/.Xdefaults` file) widget and application code options such as colors and initial values. These user-defined resources start out as ASCII values (as with the `~/.Xdefaults` file), but are converted by the Resource Manager (RM) to the appropriate internal value. Xt defines an initial set of these resource types (such as integers, colors, and booleans) but the widget developers are free to create their own types.

The biggest technical issue discussed during the TIMs was how to implement X resources in Ada/Xt. The existing C/Xt implementations have their own style for implementing a RM, but there was some feeling that this approach led to a lot of duplication. The binding developers wanted Ada/Xt's RM to avoid this. Not surprisingly, all the widgets will depend on the intrinsics package.

Because these resources have to follow the widget class and instance hierarchies, they have to fit into how Xt inherits properties. From the X end user's perspective, the `~/.Xdefaults` file can contain regular expressions that allow the user to wildcard how a resource is assigned to some or all of the widgets in an application's (dual) inheritance hierarchy. The widget developers also have to connect in these resources to their code. Consider figures 4-6 and 4-10 that show examples of widget hierarchies from the application developer and widget developer points of view. The Ada packages that define the resources and the operations on them have to be created and placed somewhere within these hierarchies.

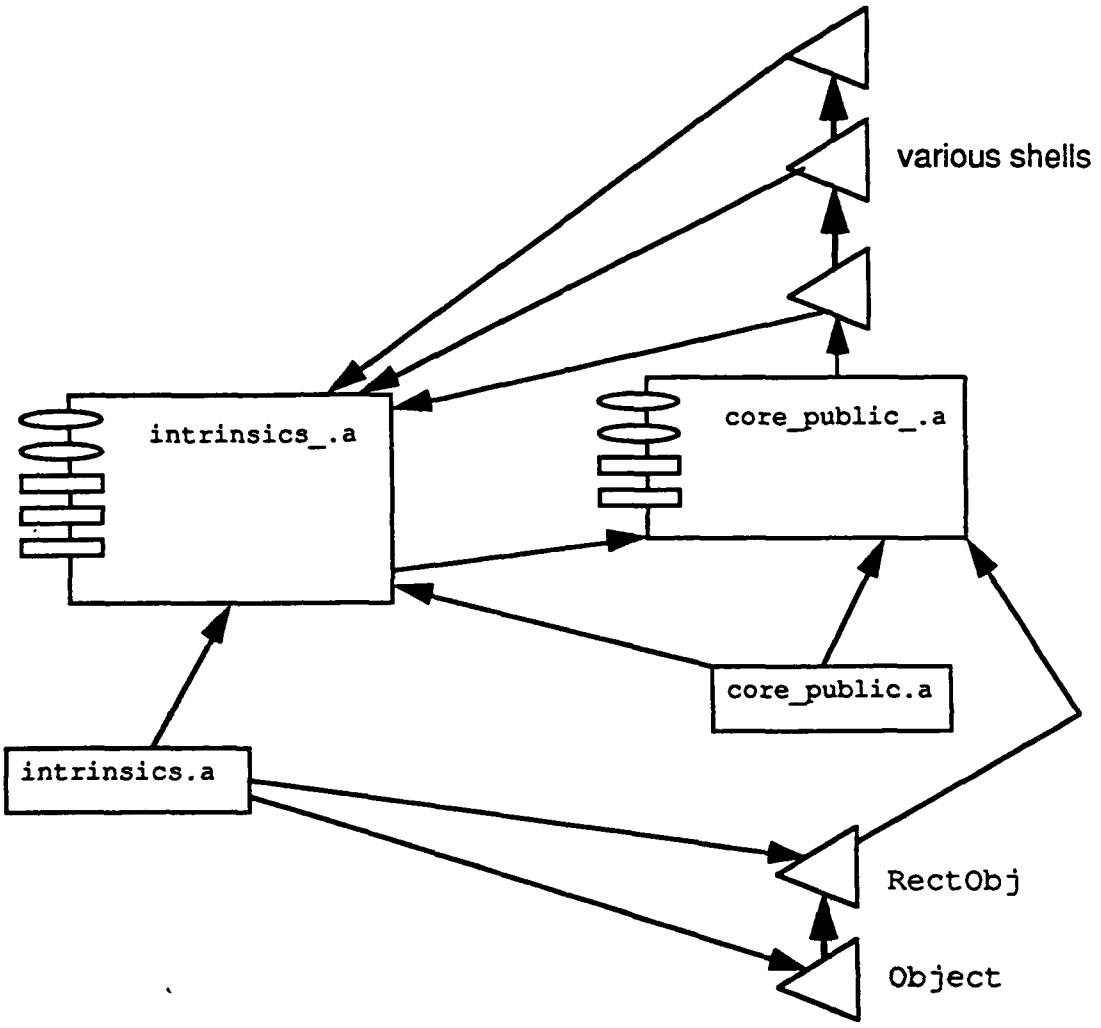

Figure 4-10. Relationship of Intrinsics to Widget Hierarchy

When a widget developer creates a new resource that is to be used in some new widget, a new Ada package can (optionally) be created that is withed into the new widget's package and any applications code using that resource. A standard base package contains all the definitions and operations on the resources defined by the base widgets. Widget developers could just bundle the new resources into the middle of the packages containing the widget. But since these resource Abstract Data Types (ADTs) will be generally useful (to future resources as well as applications code), separating the resources into separate ADTs from the

widgets allows separation of concerns. Figure 4-11 shows how a new resource ADT package might connect to the other widget packages.

In this example there are three widgets (named `core`, `simple`, and `label`) that are built on top of each other in a hierarchy, where the arrows (→) indicate the direction of who gets withed into what package. Several of these widgets use the base resources package, which is generic (indicated by the dotted or dashed lines) and provides instances of the primitive RM packages. The `label` widget and the `application` program both use a `new_resource` package that was written by a widget developer.

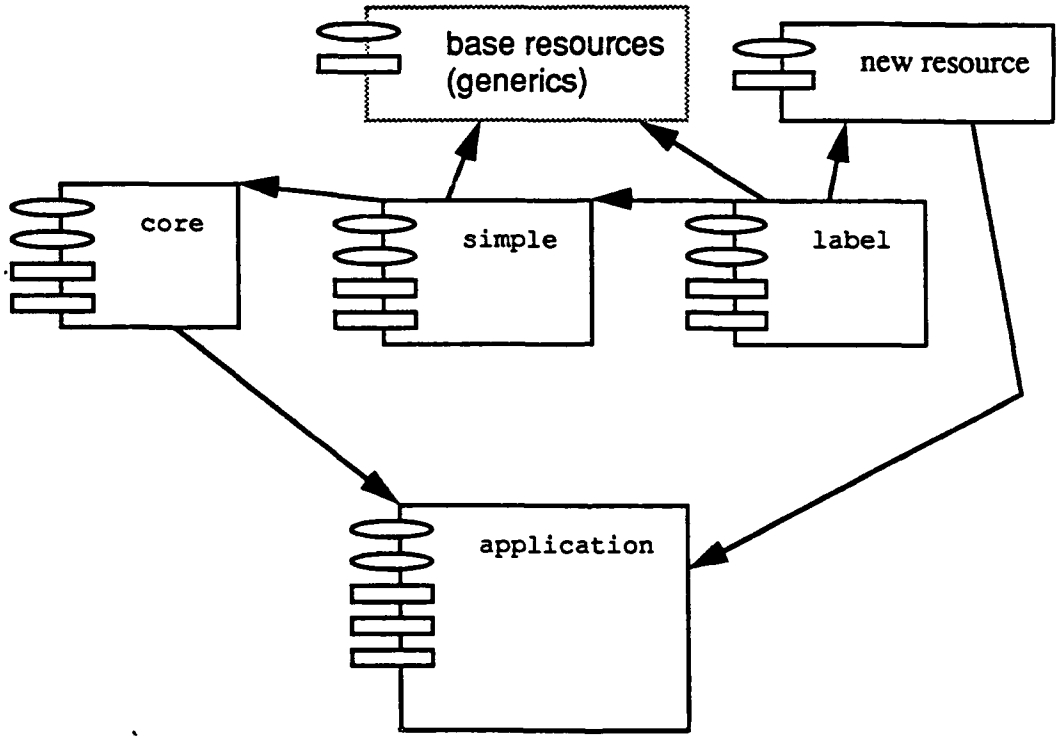

Figure 4-11. Widget and Resource ADT Relationships

One RM issue discussed was where to place these new resource packages with respect to both Ada packages and Ada source file hierarchies. These new resources could be placed into the middle of the base resource package, but that would require editing and recompiling that package (and all the other packages that use it, which could be an extensive list as figure 4-11 would imply). Placing the new resources in a separate package and separate source file was a better decision. These Ada resource source files will use a naming convention where "`..._R.a`" indicates resources. Despite some arguments that the RM and individual resource packages (source files) do not deserve to be treated like general utility programs, the binding

developers decided to follow the C/Xt conventions and place these source files under the Xmu directory (see figure 3-1 for the source directory tree).

The Ada/X binding developers wanted to build up resources and widgets (in hierarchies) for applications without having to pass system.address, because the RM packages (which have to convert or compile the ASCII ~/.Xdefaults file to a binary format) won't have access to the new resources' type definitions. They also do not want to force massive recompilations when new resources are created or debugged.

The existing C/Xt implementations use some tricks with C's untyped (and unchecked) data. With C/Xt, one passes the address and length of the place (object) that the new resource's conversion routines will eventually place the compiled resource. C allows (encourages) the allocation of untyped data of the appropriate length with the RM (who will not know the type of the data to be placed in it) for eventual inclusion with the rest of the widget instance's data.

Ada/Xt starts getting into trouble when the compiled resource takes up more than 32 bits of data. When the resource is ≤ 32 bits (as with a color value), a standard Ada integer can be allocated and be converted (with the UNCHECKED_CONVERSION function) to the appropriate type without loss of information. But with compiled data types of > 32 bits, the RM can no longer make this assumption. The binding developers debated the relative merits of passing the ADDRESS of an Ada untyped data (such as a character string or a byte array, which is what Unisys does) or always requiring strongly typed data (which is what SAIC does).

An alternative was to create a generic address type and convert an access type of a new resource (which might be a record) to an access of this type. This avoids the ugly solution of passing system.address everywhere. One question this raises is where the conversion (compilation) functions go (in the base or individual resource packages) and the type of their arguments. Each resource will have its (overloaded) set_resource subprogram, but with this new access/address type. The base conversion functions would continue to be placed in the base widget packages.

While the problem described above applies to widget developers, the application developers will have a similar problem. The Xt_Set_Arg and Xt_Get_Value subprograms are not provided by the RM but use similar mechanisms to deal with argument values that are ≤ or > 32 bits. Ada/X would like to use the same Ada/Xt implementations for both these subprograms and the RM's ones. But, there is concern that target data types involving potential dope vectors (such as the unconstrained arrays in Xt_Label) would invalidate these conversion assumptions. The resolution was to assume that the string (for example) will have a known length (will no longer be unconstrained) by the time it is passed to conversion functions such as To_Xt_Argval.

A related issue is how default resource values are set (usually embedded in the source code). Again the problem is with resources that are ≤ or > 32 bits. Those values that are ≤ 32 bits could be inserted into the middle of a static Ada array or record as integers and then converted as needed. But larger (> 32 bit) resources (that are themselves arrays or records) would need access pointers and/or some sort of address conversion function. One school of thought states that there are only a few of these complex resource types, so programmers and widget developers need to create only a few conversion functions. But another argument is

for consistency (either pass everything or nothing by address) and staying within Ada's strong typing model.

The existing C/Xt RM offers little guidance in this area. Without enumerated functions, C/Xt forces the widget writer to manually pass in the resource's format as well as its default value. C/Xt further complicates things by being inconsistent in its usage of immediate and addressed values. For example, these two C/Xt statements are equivalent:

```
set_resource(...,xtrboolean,&xt_true,...);        /* & = address of */
set_resource(...,xtrimmediate,xt_true,...);
```

After a lengthy discussion, the binding developers decided to use SAIC's approach of passing all default resource values (>32 bits) through the resource's ADDRESS attribute instead of using Unisys' approach of using generic instantiations. While generics are probably cleaner, they had problems with some classes of resources in `set_resources` and `get_resources`. The ADDRESS values will be passed as `X_Long_Integer`, since there are some odd cases where addresses such as `10` or `-1` are used as special constants. This points out just how hard it is to deal with "untyped" data in Ada.

They decided to change the `set_resources` subprograms from procedures to functions (returning `resource_records`) so these functions can be used during elaboration time to construct resource arrays on the fly. Unlike C/Xt, there is no need to pass the default resource's format (the `xtr...` arguments above) because overloaded `set_resources` Ada functions can be defined instead. Eventually the binding developers decided to overload the parameter profiles for these data types: `X_Long_Integer`, `X_Address`, `String_Pointer`, `string`, and `Xt_Default_Resource_Procs.Xt_Default_Resource_Proc`.

One area that was in question was how to handle both Unisys' elaboration-time initialization of resources and SAIC's explicit calls to compile resources. Any uncompiled (embedded) resources within the `resource_records` returned above have to be converted eventually.

The related `Xt_Set_Values` and `Xt_Get_Values` subprograms will be similar, using overloaded arguments to support `X_Address`, `String_Pointer`, `string`, and the other data types. When callback procedures (as described in appendix B) have to be passed, a generic instantiation is used to create the `proc`. Generics would also be used with certain new resource class types, such as those involving enumerated values.

As discussed earlier, new resource classes considered generally useful will be placed in their own Ada source files (..._R_.a for the specification and ..._R.a for the body), with any special conversion routines placed in files under the Xmu source directory. These conversions would apply to the STARS Ada/Xt widgets being written; future widget developers can invent their own conversions.

An issue that affects the RM as well as the rest of Ada/Xt is how to treat boolean values. As discussed earlier, Ada/Xt needs to use its own `Xt_Boolean` data type instead of the predefined `standard.boolean` type because precise control of the representation specification (for compatible record layouts) is needed. The binding developers had to decide which subprograms should naturally use `Xt_Boolean` and which should use the

regular boolean in their arguments. They decided that subprograms that deal directly with information from these records (such as `Xt_Boolean_Resources`) would use `Xt_Boolean`, while subprograms that are more naturally thought of as asking a yes/no question (such as `Is_Something_True`) would return a normal boolean. This should reduce the number of conversion functions between `Xt_Boolean` and `standard.boolean` (as when used in Ada `if` statements).

As discussed earlier, C/Xt compiles resource information "in place" (overwriting the original uncompiled data) while Ada/Xt will use different record fields to hold the uncompiled and compiled versions of a resource. The compiled versions will be placed first in the record's layout, just in case Ada/Xt has to interface directly with C/Xt widgets (which will expect to see the compiled resources at a particular place in the record). In keeping with the Ada/Xt identifier naming conventions, the data type `widget_resource_ptr` was renamed to `widget_resources`.

4.4 WIDGET TYPING AND SUBCLASSING

Another major technical issue discussed by the Ada/X binding developers was how widgets would be typed and subclassed (in the OOP sense). Part of this issue is how widgets are created. For example, could an application developer create a widget through an allocator or an access to the widget's type? Or does the programmer have to make an explicit `create` function call? This is related to the `Xt_Create_Managed_Widget` versus `Xt_App_Create_Shell` issue discussed in appendix B. Ada/Xt would like to follow the C/Xt convention of having only one subprogram that creates a shell (where special parameters to `Xt_Create_Managed_Widget` might be used to circumvent that).

SAIC's approach places a `create` subprogram in each widget's package instead of having a central `create` function in the underlying `Xt_Intrinsics` package that needs to have the `SIZE` attribute of each widget passed to it. This approach uses an allocated **new** in the Ada source code and a few simple consistency checks. Note that this approach would break if C/Xt changed to add new checks and/or calls during creation.

Unisys encapsulates widget building into a central `create` function, so any future changes (such as those for X11R4 and X11R5 intrinsics) need to be done in only one place. This is where the widget creation callback procedure ideas discussed earlier would help, since the **new** that allocates the widget's storage would be done in the widget's Ada name-space.

But all these creation `procs` would prevent the subclassing (through subtyping) of widgets. An alternative would be to use Unisys' direct calls to the C `malloc()` function to create (more) untyped data. As with the RM, there are tradeoffs between who does storage allocation and whether the creator knows exactly what type and size of data to create. An alternative discussed was to use the `extension` field of each widget, but there was resistance to the idea of using this field since others in the Xt world might also be using it. Eventually the binding developers decided to use either `malloc()` calls or allocations of untyped byte arrays.

These widget creation issues have to be considered in light of how widget classes can be subclasses and inherit information. As shown in figure 4-4, widget classes are arranged in hierarchies. So from an application developer's point of view, the widget class hierarchy shown in figure 4-10 above looks something like the Ada code fragment below.

```
...
     type widget is private;
     subtype core is widget;              -- subclassing thru Ada type
     subtype composite is core;           --  hierarchies
...
private
     type widget is access widget_record;
...
```

But as the widget class record layouts of appendix B show, each level in this hierarchy adds another layer of fields to hold data particular to this widget class. If just the subtyping mechanism above were used, how would subprograms that are passed a `core` widget get at the fields from a more complete (higher level) widget such as `composite`? Would conversion functions be needed to get at the right type? Binding developers do not want to force the application developers to do everything as a `widget`; they want to instead encourage the use of `shell`, `label`, `command`, or whatever is appropriate in the application (without making programming too difficult for the widget developers either).

Ada/Xt's developers eventually decided to go with visible subtypes (as shown above) in the package specifications, with the needed conversion and access functions that get mapped to the full record layout definitions in the package's semiprivate portion (as described earlier) that contains the real widget class definitions. Note that while this approach works with subtypes, derived types will not work as cleanly because of potential conversion functions. Any SAIC or Unisys-specific routines (specific as to the way they handle widgets) go into the semi-private packages to encapsulate Ada/Xt implementation details. Warning messages will be issued if a programmer calls an access or conversion routine that is supposed to be used only on the other Ada/Xt implementation.

The Ada source file naming convention for intrinsics widgets was decided to be:

File Name	Contents
`xt_core_.a`	`core` widget's public specification,
`xt_core.a`	`core` widget's public body, and
`xt_core_p_.a`	`core` widget's semiprivate specification.

The abbreviation of private to `p` avoids (or at least limits) the problems of lengthy widget names (such as `Xt_Application`) resulting in file names that are too long for some host operating systems. Note that non-intrinsics widgets (such as `label`) would not have the `xt_` prefix in front of their file names; instead the source file name would be `label.a`.

SECTION 5

FUTURE DEVELOPMENT WORK

The last of the TIMs brought to an end the formal review and technical exchange portion of the STARS task. Some open issues remain (besides implementing and documenting the final software products) that both the binding developers and STARS management should consider for the future. The X Window System, any language bindings to it, the commercial (consortia) widget sets sold for it, and even the Ada language that programmers intend to use with it, are all scheduled for change during the 1990s. The STARS program, commercial interests, and other interested parties need to begin planning about what to do next with Ada/X.

SAIC and Unisys have created a tutorial on their Ada/X work that was first presented at the January 1991 Fifth X Technical Conference in Boston, MA. This tutorial was based on some of the earlier talks they have given, such as at the recent Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda) Summer 1990 Conference. This tutorial assumed that the attendees had a working knowledge of Xt, so the talks concentrated on how Ada/Xt is used and how it makes Ada applications programming easier by hiding many of the messier details of Ada/Xlib. Beyond this tutorial and the final Ada/X documentation to be delivered by the two developers, there are no announced plans for application developer or widget developer documentation or instructional/training materials. Given the large amounts of instructional and reference material that has been written for C/X, similar levels of documentation for Ada/X may be required for general acceptance.

The STARS task produced an Ada/Xlib binding to the X11R4 release of the X Window System and an Ada/Xt implementation to X11R3. The Athena consortium is well on its way to preparing X11R5 (and then eventually X11R6), as more features and improvements are added to the reference implementation of X. Beyond the hope that the Ada/X work transitions quickly to commercial Ada compiler and binding reseller concerns, there are no real plans for handling future upgrades to X. Transitioning the control of this language binding standard to the appropriate standards group for formal definition and periodic upgrades remains to be done.

The STARS policy toward commercial widget sets such as Motif and OL remains undecided. With application developers tending to use one widget set or the other to avoid writing a lot of widgets from scratch (instead, one just reuses and/or subclasses from the widgets available), Ada application developers will expect bindings or implementations of one or both of these widget sets. The decision of when and which widget set(s) to implement rests with STARS, commercial binding developers, and/or with the consortia themselves.

This and other decisions will depend on what the STARS direction for the future will be. As STARS holds more workshops and other feedback sessions, there may be decisions that affect ongoing (or previous) STARS products such as this Ada/X work. For example, the impending upgrade to the Ada language (Ada 9X, resulting in MIL-STD-1815B) may provide language features that COTS package interfaces (such as databases and this Ada/X

work) will want to exploit. Already Ada 9X has announced plans to explore an Ada/X binding for the new version of Ada, to see just how improved the 9X OOD/OOP features are. The relationship of the STARS technology work (such as Ada/X) with all the other groups developing future Ada technology remains to be worked out.

Another future development area is the creation of widget sets for specific application domains. Most current widgets are for very general use. A particular application domain will want (reusable) widgets that capture both the user interface and behavioral semantics of a set of applications. A related development area is the creation of widgets, Ada/X libraries, and client/server relationships that are usable in time-critical applications. Plenty of opportunities remain for binding developers, UIMS writers, widget developers and application developers in this area.

SECTION 6

POTENTIAL STANDARDIZATION

The STARS Ada/X binding developers agreed that there needs to be some sort of standards group responsible for just the Ada bindings to the rest of X. As STARS and other organizations listed in table 2-1 generate Ada/Xlib, Ada/Xt and other specifications, they will need to transition these specifications to a standards group to be maintained. The initially incompatible Ada/X systems from just within the STARS program are examples of what can go wrong for the entire Ada/X community if too many incompatible bindings are developed. The X Window System levels that this STARS Ada/X task tried to bind to (Xlib, Xt, widget sets, and UIMSs as outlined in figure 2-1), are all in a state of flux as new technical improvements are continuously being added. Each of these changes might be implemented differently by the binding developers, hurting overall Ada/X portability. These changes to C/X (at least) will be controlled by a series of standards groups; some are American National Standards Institute (ANSI) groups (such as X3H3) and some are Institute of Electrical and Electronic Engineers (IEEE) groups, such as the Portable Operating System Interface to UNIX (POSIX) committee P1201.

Currently, there are no clear plans on how (or who) to set up such an Ada/X standards group, or how that group might be related to the overall STARS effort in this area. There appears to be some time left before major decisions about Ada/X binding standards groups have to be made. But, it may be time to start laying some of the groundwork for such an effort, and starting to build interest both inside and outside of STARS based on the competitive quality products that (hopefully) will come out of this task and/or the commercialization follow-ons.

Any standardization work has to start with something; in this Ada/X binding work, the initial base products and specifications that will form version 0 of any standards could come out of this STARS task. So before a standardization group formally begins its work, it would be a good idea for the products of this task and some of the other Ada/X developments to be completed so the standards get off to a running start. Since this STARS task produced deliverables that will jump start a major Ada industry standardization effort, it is important that this STARS task get some external technical feedback to begin building some broad consensus (and catch any major design errors early) beyond the small membership of the STARS binding developers. This is a reason to continue communicating with the broad Ada community as in upcoming SIGAda conferences.

Any Ada standardization group would be within the structure of an existing standards group; there is no point in creating some all new group just for Ada bindings. This means that the Ada group would have to conform to practices being used by its related groups. For example, the IEEE POSIX effort already has a committee (P1201) looking specifically at creating industry standards for X, so P1201 could be a good place to form a subcommittee for

dealing just with Ada bindings. POSIX already has a subcommittee that deals with just Ada binding standards to the base operating system (P1003.5), so language-specific bindings are not new to POSIX.

One issue to consider in the timing of the formation of any Ada/X binding standardization group is the progress being made by the related standards groups. In POSIX, the trend is toward the development first of language-independent standards for the functionality of some system, with language-specific bindings then developed on top of this standard. For X, this trend leads to some confusion because there are different groups working on different standards (at different rates of completion) for the various layers of X.

At the base of the X Window System is the X protocol defining the bits and bytes that are passed between client and server (see figures 2-1 and 2-2 for overviews). The ANSI X3H3.6 subcommittee is almost done with this standard. Since this protocol layer is well below what any Ada application program would have to deal with (with the possible exception of the Rational Ada/Xlib implementation) and since Ada/X generally abstracts away this layer by only dealing with the X Window System through higher layers, there may be no need for an Ada-specific binding standard at the protocol layer.

The next layer up is Xlib. The development of the functionality (specification of behavior) is done by the Athena Consortium itself, with the proposed IEEE P1201.4 group responsible for creating the formal standard. A problem has arisen here in that the Athena Consortium wants to hold off any standardization (in C, Ada, or anything else) at the Xlib layer until they have come up with a solution to the tricky problem of internationalization (the support of non-Roman alphabets such as Kanji, Hangul, Arabic, and Hebrew) in X11R5.

One product that came out of this STARS task was the common SAIC/Unisys Ada/Xlib binding, yet it is not clear that P1201.4 will have a language-independent Xlib specification for this Ada binding to claim conformance to. It is quite possible (and likely) that the products that come out of STARS and/or a commercial concern could become *de facto* standards that serve the Ada community's need until formal IEEE POSIX standards become available. But in the short-term, there could be problems in specifying conformance to standards that really do not exist yet.

When moving above Xlib, an application developer moves into X toolkits, widget sets, and controversy. While the standardization community has settled on using the same Xt intrinsics that this STARS task developed Ada implementations of, the widget sets and "look and feel" issues remain highly competitive. P1201.1 is trying again with a very high level abstract interface known as the Virtual Application Programming Interface (VAPI) that was developed for the XVT commercial virtual toolkit that claims conformance to almost any windowing system (including Motif and OL).

All this has greatly slowed the efforts by P1201.1 to produce a formal standard. MITRE's experience in this area is that standards take at least two years to get through the review and balloting process. The outlook for the Xlib layer standard is relatively hopeful. Assuming that the Athena Consortium can remain on schedule in producing an X11R5 in 1991 that addresses the internationalization issue so P1201.4 can have a language-independent

specification of Xlib ready to go to standard, a related Ada/Xlib binding standards committee (perhaps numbered P1201.5) could be formed. But the outlook for Xt standards remains clouded.

There are ways to speed up the (nominally two-year) standardization effort if preparations and consensus are built ahead of time. If STARS used the time between now and the release of X11R5 in 1991 to get Ada industry feedback on the evolving Ada bindings, then the balloting step could be moved up. A new P1201 group could go almost straight to balloting, saving nearly a year in the standardization effort. Note that there are some risks in early solicitation of industry comments. The STARS Ada/X binding may have made good progress just because the number of participants was small.

The Ada/X community will have to address the widget set issue at some point. The X application development community seems to be lining up behind commercial widget sets such as Motif's and OL's, with older and smaller widget sets such as the Athena and Xray sets falling into disuse. In addition to the size of these commercial widget sets (Ada bindings and/or implementations would be a major undertaking), there are difficult copyright and commercial advantage issues to consider. Do government programs such as STARS want to become involved with one or both of these industrial consortia?

Another standardization issue to consider is the moving targets being followed. The original STARS Foundation's Ada bindings were to X11R2; the initial releases of both the SAIC and Unisys Ada bindings are to X11R3; the final STARS Ada/Xlib bindings are to X11R4; any P1201.5 standardization work in future years would have to deal with the coming X11R5; and there are already plans for X11R6 (that will include threads, where Ada tasking can really make an impact). The commercial widget sets are also moving targets (such as Motif versions 1.1, 1.2, 1.3, 2.0, etc.). Note that the commercial X vendors don't always release their products in synchronization with the latest Athena Consortia releases (Motif remained at X11R3 for a while after X11R4 was released). This means that Ada/X binding development will remain an ongoing process, requiring a consistent source of support for both the programming and standards maintenance.

With confusion and competition reigning at the widget set layer, STARS might want to look toward the higher UIMS layer for standards that are also of use to application developers. POSIX P1201.3 was looking at standards for this area before this effort collapsed. Serpent and TAE+ are examples of UIMSs that handle Ada APIs. An issue for Ada/X binding developers to consider is whether the Ada/Xlib and Ada/Xt interfaces can handle the Ada source code generated by these UIMSs, particularly given the Ada host compiler problems noted by application developers using machine-generated Ada code.

Another question that Ada/X binding developers should consider is how they should deal with the recent National Institute of Science and Technology (NIST) Federal Information Processing Standard (FIPS) on user interface models (FIPS 158). NIST has defined a seven-layer reference model (patterned after the Open Systems Interconnection [OSI] network reference model) that is used to organize standards. FIPS 158 defines a user interface reference model that is closely patterned after the X Window System's architecture stack (see figure 2-1).

FIPS 158 Model	X11R4 Reality
application	application
dialog	UIMS
presentation	?
toolkit	widgets (maybe)
subroutine foundation	Xt intrinsics
subroutines	Xlib
byte stream	X protocol

Note that while FIPS 158 is promoted as a reference model, it is only very loosely connected to other NIST reference models such as the one being created for Ada Programming Support Environments (APSE). The integration of COTS (as with an X11 package) and application models of behavior (as in using a common client/server model) is more complicated than many application developers realize. Also note that FIPS 158 avoids the Motif/OL question by leaving the widget set issue open.

It is possible that FIPS 158 will have little, if any, impact on Ada/X bindings because the FIPS' reuse of the X architecture stack helps ensure that whatever products come out of the binding, developer will automatically conform with the spirit of this FIPS. Experience with other FIPS have shown them to be mainly placeholders for other standards. The FIPS dealing with the POSIX operating system specification (being done by P1003) even contains a forward reference: the POSIX P1003 standards (and there will eventually be many of them) have not gone through the formal balloting and approval steps yet; this has not prevented NIST from issuing FIPSs requiring conformance to draft standards.

These FIPSs also seem to be automatically upgraded as whatever real standards they point to are upgraded. So if the current FIPS 158 contains a reference to X11R3, as X11R4 and later releases are done, then they become effective within the FIPS. An issue that FIPS 158 really does not address is standards skew as different organizations upgrade their work. Earlier, this report noted how the higher level widget sets sold by one organization may not be the latest X release (as was the case with Motif). Upgrading the bindings to specific languages such as Ada could also fall behind the release of an upgrade to a base specification. Ada/X standardization efforts should keep the evaluation of related standards such as FIPS 158 in mind as Ada/X matures.

Another standardization issue to consider for FIPS 158 and Ada bindings is the extensions being made to the X Window System by various developers and researchers. For example, the Video Extensions to X (VEX) and the Programmer's Hierarchical Interface for Graphics Systems (PHIGS) Extensions to X (PEX) both define functionality that some application developers will need but which lies outside the traditional X Window System reference modelAt some point, Ada application developers might want bindings to extensions such as PEX and VEX.

SECTION 7

DISTRIBUTION ISSUES

One issue addressed during the STARS Ada/X development was the distribution of products in general and the software (source code) for the Ada/X binding in particular. There were a variety of complaints expressed by the Ada community about the distribution of the earlier Ada/Xlib binding developed under the STARS Foundation's work. No one seemed to be in charge of collecting and distributing the bug notices and fixes for this early product. Eventually, commercial companies (such as GHG) would turn this problem into a sales opportunity, but the delay between initial release and commercial support was still troubling.

One reason the X Window System has gained its popularity is the ease by which the software (particularly the base or reference version) can be obtained. The source code is publicly available by both magnetic tape and FTP (File Transfer Protocol, a direct computer-computer network connection) distribution means. The STARS Ada/X binding developers received permission to distribute the software via FTP access as well, to aid in quick distribution.

The Unisys version of the completed Ada/X software is available via `anonymous` FTP from the STARS repository on `STARS.rosslyn.unisys.com`. The SAIC version is available via magnetic tape from the STARS office at Boeing. Other sites such as the `grebyn.com` computer have made this software available for FTP access.

The eventual application developers who use these bindings will need opportunities to sit down with each other and the binding developers to provide feedback on binding use. This is especially true as Ada/X is used with new Ada compilers of modified versions or existing reference compilers. Some of this can be done electronically, such as through the existing `x-ada@expo.lcs.mit.edu` mailing list maintained by the Athena Consortium.

One distribution issue that proved to be more difficult than originally expected concerned the file and unit "headers" that are placed in the source code. A final header scheme needed a resolution of exactly which copyright notices (for Athena, STARS, etc.) should be placed into the different files. Complicating this were the different copyright notices that the Ada/Xlib source code (which is a thin shell around copyrighted Athena C code) and the Ada/Xt source (developed under STARS funding from scratch to conform to Xt intrinsics behavioral specification) would need. If the choice of which (commercial) widget set (such as Motif) is resolved, then another set of copyright notices for the widget source files may need to be created.

SECTION 8

ACQUISITION GUIDANCE

This paper has reviewed the progress made by the STARS Ada/X binding developers and presented a high-level overview of their products. Besides capturing lessons for future application developers, widget developers, UIMS writers, and binding developers, this paper tried to give some guidance to acquisition support people who must review or guide the efforts of others.

One point is that the X Window System is a very complex set of interrelated products and interfaces (as figures 2-1 and 2-2 implied). Acquisition documents containing language such as "*shall* use X Windowing System" are of little help to developers because of the many levels of interfaces the programmers may choose to use for supporting different parts of an application. Even language such as "*shall* use a binding to X" is not helpful; as discussed above, there have been products that provide full Ada implementations of parts of X and so requiring a binding is over-specification.

Another point is the rapidly shifting nature of X. As discussed above, products and bindings can conform to a variety of X standards such as X11R3, X11R4, and X11R5. Commercial tools, products, and widget sets may be up to date or lag behind these base Athena Consortium releases. Formal (ANSI and/or IEEE) standardization is a rapidly developing and evolving area. Acquisition people should be very precise as to the release number, versions, and product dates of subsystems and bindings that any (Ada) applications code must interface to, *if* that is an important requirement to a program. Otherwise the acquisition staff should not over specify a latest version number that is likely to change in the near future. If the X Window System is an important part of a system's software, then the project should keep up to date with technical and commercial developments in this area.

Acquisition support people should also be aware of the tradeoffs between design abstractions, run-time performance, software portability, and other factors. As figure 4-6 showed, the design of an application using X can rapidly become complex. The software architecture and flow of data/control, perhaps once represented entirely within applications code, now must include COTS behavior as provided by X intrinsics and widget sets. OOD concepts such as subclassing and inheritance provide new paradigms for the application developer to exploit, but require that the acquisition people review the software from a different perspective. Traditional deliverables as called for in DOD-STD-2167A (DOD, 88) may not be as applicable to applications using X's architectures.

Reusability and COTS integration are not free; the complex architectures shown in figure 4-6 impose at least some run-time performance penalty over custom-written windowing systems. Systems with strict performance requirements should be careful in how they benchmark and track performance since so many parts of X (and Ada bindings to it)

contribute to overall performance. Acquisition support people must also balance the relative ease of programming and portability of Ada/X at the higher levels of figure 2-1 with the run-time overhead of dealing with more abstract layers.

LIST OF REFERENCES

Buhr, R. J. A., 1984, *System Design With Ada*, Prentice-Hall.

Byrnes, C., 1989, "A DIANA Query Language for the Analysis of Ada Software," *Proceedings of the Seventh National Conference on Ada Technology*, U. S. Army Communications – Electronic Command.

Byrnes, C., 1990, "Formal Design Methods for Dynamic Ada Architectures," *Proceedings of the Eighth National Conference on Ada Technology*, U. S. Army Communications – Electronic Command.

Department of Defense, Ada Joint Program Office, 1983, *Ada Language Reference Manual*, ANSI/MIL-STD-1815A.

Department of Defense, Joint Logistics Commanders, 1988, *Defense Software Development Standard*, DOD-STD-2167A.

Emery, D., 1990, "A Prototype Implementation of the Ada Binding to POSIX," *Proceedings of the TRI-Ada '90 Conference*, Association for Computing Machinery.

Evans, A, and K. Butler, 1983, *Descriptive Intermediate Attributed Notation for Ada Reference Manual*, TL-83-4, Tartan Labs.

Latour, L., 1990, "A Methodology for the Design of Reuse Engineered Ada Components," *Proceedings of the First International Symposium on Environments and Tools for Ada*, Association for Computing Machinery.

Lewin, S., 1989, "Ada Implementation of an X Window System Server," *Proceedings of the TRI-Ada '89 Conference*, Association for Computing Machinery.

Luckham, D., et al, 1987, *Anna: A Language for Annotating Ada Programs*, Lecture Notes in Computer Science #260, Springer-Verlag.

Scheifler, R., J. Gettys, and R. Newman, 1988, *X Window System: C Library and Protocol Reference*, DEC Press.

SEI, 1989, *Serpent Overview*, ESD-TR-89-08, Software Engineering Institute.

Szczur, M., 1990, "The Transportable Applications Environment Plus (TAE+): A User Interface Development Tool for Building X Window-based Applications," *Proceedings of the Fourth X Technical Conference*, MIT Athena Consortium,.

Wallnau, K., 1990, *UR20 – Process/Environment Integration Ada/Xt Architecture: Design Report*, STARS-RC-01000/001/00, STARS CDRL 01000, AD-A228-827.

APPENDIX A

DETAILED ADA/XLIB CHANGES

This appendix lists the detailed technical issues related to an Ada/Xlib binding that were identified and the resolutions on which the STARS binding developers decided. The listing of resolutions and rationales are Tim Schreyer's, with some additional commentary to give a flavor of what an Ada application developer might say.

A.1 RENAME THE MAIN INTERFACE PACKAGE X_WINDOWS TO X_LIB

The rationale for renaming the main package of Ada/Xlib was that it would improve the way people think about and refer to the bindings and make Ada/Xlib a closer fit as a binding to Athena's C/Xlib. Automated support for existing applications would make this change relatively painless.

There was concern that this name change would cause heartburn to the existing users of Ada/Xlib bindings, going all the way back to the original STARS Foundation's binding. The developers felt that a UNIX `sed` script and/or `emacs` macro could automatically make the binding change painless. This was an example of the sort of change that needs to be made now before the world assumes that package `X_Windows` means all of X instead of just Xlib.

One issue the single interface package raises is the name-space management issues caused by a monolithic Ada/Xlib binding package. Some who have looked at the Ada/Xlib binding were worried about the huge name-space (all those Xlib subprograms, packages, and types) that suddenly become visible to the application when a `with X_Lib;` is done. The binding developers are hopeful that a smart Ada linker will silently eliminate all the dead Xlib which is visible but never used in an application; given the Ada compiler problems noted during these TIMs, that may be wishful thinking. The ideal solution would be to separate the gigantic `X_Lib` package into its major (currently nested) subpackages. Both SAIC and Unisys have attempted this, with the attempts failing because so many of the Xlib (nested) packages have intertwined dependencies that a linear compilation order couldn't be found. Some Ada application developers will not be happy with the idea of having to fully qualify all the names of the nested (at two levels) `X_Lib` packages/subprograms and/or having to use the dreaded Ada `use` statement as a shortcut.

A.2 CHANGE THE TYPE OF X_LIB.EVENTS.EVENT_TYPE

This would change the type from an Ada enumeration type to a `new Long`-derived type of an integer type (`Long` is a 32 bit integer type in Ada/Xlib). The rationale for this is to allow additional event types to be added with ease to the implementation at the Ada/Xlib level and above.

A.3 CHANGE THE BASE TYPE FOR MASKS IN ADA/XLIB

The change would be from boolean array subtypes to private 32 bit integers. Ada/Xlib provides mask type operations like `and` and `or` in a visible package. The binding developers defined a base `Mask_Type` as a `new long` (32 bit integer). This can be subtyped for specific mask types. They also defined an operations package for `Mask_Type` and determined the location and visibility of the `Mask_Type`.

The rationale for this was because attempts to implement masks as boolean arrays caused difficult problems in a binding like Ada/Xlib where masks travel frequently through the interface. This change will improve Ada/Xlib and higher interface level performance by removing expensive conversions between boolean arrays and integers and elaboration overhead for boolean arrays. In addition, this change will circumvent many compiler checks. A potential problem concerns derived types, being used here to encapsulate Xlib masks. Consider the following code fragment, where the lines in underline indicate areas of trouble:

```
package X_Lib is
     ...
     type x_mask_type is private;                -- trouble below
     type x_mask_type is new long;               -- forced to use instead
     ...
     package events is                           -- nested package
     type event_mask_type is new x_mask_type;    -- illegal
     ...
     subtype event_mask_type is x_mask_type;     -- O.K., but
     key_press_mask : event_mask_type := 2;      -- illegal private assign
     ...
     type event_mask_type is private;            -- again O.K., but
     key_press_mask : event_mask_type := 2;      -- illegal private assign
     ...
     type event_mask_type is new x_mask_type;    -- works with derived type
     ...
     private
     type event_mask_type is new x_mask_type;    -- doesn't solve assign
     ...                                         -- to key_press_mask
     end events;                                 -- problem
end X_Lib;
```

Figure A-1. Problems with Mask Data Types

The problem described above also shows up with types `drawable`, `context`, and `X_Id`, where these must be defined as `new card32` instead of `private` because of some direct assignments and conversions that are done.

A.4 EXAMINE PACKAGING FOR OPTIMAL VISIBILITY AND COMPILER INDEPENDENCE

The developers defined two new configuration dependent packages, containing

a) `Configuration_Dependent`:
-- base numeric types and size constants
-- system configuration constants
-- a byte of the Ada null access type (`null_byte`)
-- a zero `System.Address` (`null_address`)
b) `System_Utilities`:
-- command line argument interface
-- C ↔ Ada string interface

The rationale for this change was that besides providing compiler independence by keeping compiler dependent code in the bodies of these packages, this would minimize the number of small dependent packages for Unisys, and move dependent information from bindings packages to separate packages for SAIC.

The goal here was to straighten out the tangled mess of low-level packages that SAIC and Unisys developed to encapsulate the compiler-dependent and operating system-dependent features of an Ada/Xlib binding. This was particularly for tricky Ada representation specifications.. There were other problems caused by different compiler interpretations of **pragma** `Interface`, command-line argument passing, and Alsys' use of a predefined `System_Environment` package.

A.5 DETERMINE THE ROLE OF EXCEPTIONS IN THE ADA/XLIB BINDING

The STARS Foundations bindings had some Ada exceptions sitting unused in various package specifications. The resolution of what to do with these exceptions was to remove the unused exceptions from the specification of the `X_Lib` package. The rationale was that these exceptions represent errors that occur as a result of X protocol errors, after the flow of control has passed through the bindings to the C/Xlib code. As such, they are handled by the default C error handler and are not passed back to the Ada scope. The exceptions that now exist in the `X_Lib` package are unused and should be removed to prevent users from implementing exception handlers for exceptions that cannot/will not be raised.

Error handling is tricky in Xlib; so it is hard to come up with a clean Ada binding that handles errors. For example, as an application sends commands to the X client, the lowest level (the X protocol) might delay the actual transmission over the network until some ideal cache or buffer state is reached. As result, if those commands contained an Xlib error, the error would not be reported to the application (server) until after several other legal Xlib commands were issued. So even if the Ada/Xlib binding decided to raise these (deleted) exceptions, the exceptions might be raised in the scope of some subprogram that did not even

issue the offending commands and might not have a clue of how to recover from the exception. There might be ways around this, such as passing an Ada error handling procedure to the binding (perhaps as a generic parameter), or having a distinguished Ada task for handling and reporting these Xlib (X protocol) errors.

A.6 DECIDE WHICH TYPES SHOULD BE PRIVATE AND WHICH NEED TO BE PUBLIC

Both of the binding developers had their own ideas as to what should or should not be a private type. They had to decide on a case-by-case basis which types should be made public or private. The rationale for this was that during the development of the Ada/Xlib bindings, certain types that were originally private were made nonprivate so higher level interfaces could access their structure. Although hiding of types is desirable in Ada/Xlib, some types must be nonprivate so that layers based on Ada/Xlib have a suitable interface. This was one of those clean ADT versus efficient implementation arguments. The STARS Ada/X binding developers leaned towards the clean ADT approaches, which should help in presenting a sound binding for standardization.

A.7 DECIDE WHICH CONSTANTS NEED TO BE MADE INTO OBJECTS FOR ADDRESS RESOLUTION

The developers examined and incorporated both teams' changes from constants to objects, and on a case by case basis decided if additional constants should become objects. The rationale for this was that Ada allows compilers to use registers to store constants. Some constants in Ada/Xlib binding need `'address` resolution at run-time to be used in the X RM. On a selective basis, changing these constants to objects will allow the desired address resolution.

Another discussion concerned how to best represent the base C/Xlib use of enumerated values. In some cases C uses its `enum` construct, while in other cases explicit `#define` constants are used. Ideally these would be represented in Ada/Xlib with Ada's enumerated types. But a problem will quickly arise in Ada/Xt, where the original "base" collection of some enumerated types in Xlib has to be extended with additional values that are used only within Xt. That is why C/Xlib uses `#define` constants - so C/Xt code can create new constants by increments from the last `#defined` one. Ada/Xt cannot use this trick because one cannot extend an Ada enumerated type after it has been defined. One also does not want to force the new Ada/Xt-specific enumerated values down into the Ada/Xlib type definition level. Ada `private` types will also have problems when trying to extend the values.

This overall problem is similar to the `X_Lib.Event` issue raised in section A.2 above. `Events` are allowed to be extended by the application programmer, so there was no way to statically determine in the binding what the range of event numbers would be (that is why they become integers instead of enumerations). A workaround for this would be to define a convenience function that Ada/Xt programmers could use to convert the C/Xt `#define` constants into Ada deferred constants. The code fragments below show how this function would be used. First there would be an overloaded function such as:

```
function create_constant(X: long) return X_Mode;
                                        -- where X_Mode would vary

package X_Lib is                        -- where base of type defined
...
  type X_Mode is private;               --so user doesn't play with it
  input: constant X_Mode;               -- a deferred constant
  output: constant X_Mode;              -- another deferred constant
...
private
  type X_Mode is new long;              -- treated as another number
  input: constant X_Mode := 0;
  output: constant X_Mode := 1;
end X_Lib;

package Xt is                           -- now extend X_Mode data type
...
  input_output: constant X_Lib.X_Mode := create_constant(4);
end Xt;                                 -- woe to incorrect values here!
```

Figure A-2. Examples of Constant Object Definitions

Another "convenience" function that's needed is:

```
function to_Ada_boolean(val: long) return boolean;
```

because there are C/Xlib functions that return hidden status values in the return codes. For example, a return value of `-1` or `-2` might both indicate that the function call failed, but for different reasons. The Ada/Xlib function has to get a true boolean value on its type `boolean` so that `constraint_errors` is not raised.

A.8 STANDARDIZE THE X RESOURCE INTERFACE FOR ADA/XLIB

An analysis of each implementation of the Xlib RM was done to complete a common interface for Xlib resources and resource types. The rationale was that the original implementation of Ada/Xlib did not include bindings to the resource portion of Xlib. Therefore, both SAIC and Unisys have developed resource managers that must be merged to provide a standard interface to all the Ada/Xlib bindings.

Note that the Xlib `resource_mgr` package is an additional burden to implement because the original C binding took some abuses with C's pointer-passing mechanisms. Some of the Ada

binding problems caused by trying to duplicate C's pointer and array passing can be traced to this package. The binding developers created a resource package that is more Ada-like in passing actual array back and forth, instead of forcing the use of **access** types for everything.

A potential problem with the Xlib-level X resource manager (Xrm) was resolved when a code inspection revealed that there were no instances of applications code needing to see or access the internal `hash_bucket` types within package `resource_mgr`. That eliminated the need to make that type visible and to worry about both the Ada and C sides allocating records that need to be reclaimed. A related Xrm issue was how to treat the "arrays of records" (such as user-defined `search_lists`) that are passed around the software. This turned out to be another instance of the `string` passing problem discussed above. The "simple" solution is to pass these lists around with `system.address` parameters; but this could fail at run-time if the records (particularly variant records) used dope vectors to control the record instance. In this case the **access** `search_lists` is not equal to `system.address` because the `address` will be to the start of the dope vector or record descriptor, not the start of the actual record data. Improper representation specifications can also lead to memory leakages as an application program runs.

A convention for dealing with arrays and records with embedded pointers was defined as:

records with pointers	use `record.all'ADDRESS`, and
arrays with (embedded) pointers	use `array'FIRST'ADDRESS`

Note that if C's `NULL` address/pointer is present, then the Ada/Xlib constant `zero_address` would be used. A check that Ada compilers (particularly those that do not assume Ada's `null` = 0) can accept this was done.

One question raised about memory leaks was whether an Ada/Xlib binding should add new Ada procedures (above the standard strict binding routines) to allow the declaration of C data structures (such as linked lists). There are already some new Ada procedures that are not in a strict or simplistic Ada/Xlib binding; so these new deallocation or `free` procedures would not represent a major change. A single (overloaded) `free` procedure could deallocate Ada as well as C structures. The binding developers decided that calls to `free` would be added on a case-by-case basis.

A.9 STANDARDIZE THE NAMES OF VISIBLE TYPES, SUBROUTINES, AND PACKAGES

The developers converted all procedural interfaces to Xlib style names. The rationale was that renaming the visible interface of Ada/Xlib would make the bindings a better match to C/Xlib, and a more worthy candidate for standardization. Renaming would also allow the easy reuse of the large existing body of Xlib documentation for C to be used as a functional reference for Ada/Xlib.

Some of the Ada/Xlib subprograms had an "`x_`" prefix before the name, while others did not. The same thing occurred at the Ada/Xt binding level, where some but not all subprograms

and packages had an "Xt_" prefix. Even worse, SAIC and Unisys had different naming standards for who did and did not get these prefixes. While this will be a fairly major change (at least at the cosmetic level), there was agreement that an Ada/Xlib and Ada/Xt binding standard would never get approved with an inconsistent naming standard. So the binding developers decided to bite the bullet now and change over to a consistent scheme. As with the earlier change from `X_Windows` to `X_Lib`, much of this change could be done automatically with the appropriate UNIX `sed` script.

A.10 HAVE LESS EMPHASIS ON BACKWARD COMPATIBILITY

The development strategy was to concentrate on producing the best and most lasting Ada/Xlib binding, and provide a measure of automated support for upgrading the source code of existing applications. The rationale was that the convergence of SAIC's and Unisys' Ada/Xlib produced a more stable and tighter binding to Xlib than many existing applications have. The number of existing applications is now at the lowest that can be expected. The production of an Ada/Xlib binding of the highest quality is important for the goals of standardization.

This means dropping support for an Ada/Xlib binding for X11R3 and going straight to X11R4. The rest of the world is going to X11R4, so there is no reason for the Ada binding to be left behind. As explained below, the impact to the remaining X11R3 users should be minimal through the availability of dummy X11R4-specific interface routines.

A.11 DEFINE INTERFACE TO COMMAND LINE ARGUMENTS AT THE ADA/XLIB BINDINGS LEVEL

The command line argument interface is compiler dependent and should be separated from the bulk of the bindings. The POSIX P1003.5 people have a standard interface to command-line arguments; future binding developers should look at using that interface (along with a lot of other good ideas in the P1003.5 Ada binding) to minimize duplication of effort.

A.12 DEFINE ONE REPRESENTATION OF THE MANY X EVENT STRUCTURES IN ADA/XLIB

After some discussions, SAIC and Unisys decided to keep the existing two different representations. The rationale was that currently one representation is kept for the Ada interface to the user, and the other for the interface to C through the bindings. Changing to one representation is an implementation task (not affecting the common specification) beyond the scope of this effort.

The feeling was that the Xlib event structure details would be hidden in the bodies of the various Xlib packages, so Ada application developers would never see that there are

differences in the implementation. Both development teams thought events were a complex enough issue to leave to compiler-specific maintainers. Hopefully, details of this are so low-level that they will not affect standardization.

The Ada/Xlib binding should not assume that Ada's **access** `string` is the same as C's "`char *`". The binding instead assumed that all those **pragma** `Interface (C)` routines that are called return the only safe assumption for a value: `System.address`. If this address happens to be the start of a C string `(char *)`, then one should use a conversion function on the Ada side to convince the Ada compiler that the address can be treated as the start of the "data" portion of the (Ada) array. A portability issue is how different C compilers (such as cc and gcc) would generate the layout of the records that Ada/X source code has access through representation specifications. Ada compilers (and this Xlib binding should support them all) can chose to implement `access` in a variety of ways. While with some compilers (such as Verdix) the `access` value is indeed the pointer to the start of the real data, other Ada compilers (such as Meridian, especially with an access to an unconstrained array) will return pointers to allocated "dope vectors" with the actual data pointer located somewhere else. The STARS Foundation's Xlib binding will be very nonportable across validated Ada compilers, and will therefore run into a lot of opposition if someone tries to turn it into a standard.

Our standard workaround [described in (Emery, 1990)] is to define a function like this, with the bold line below indicating where the connection to the string is made:

```
function address_to_string(pointer: system.address) return string
is
     result : string(1 .. Xstrlen(pointer));
     for result use at pointer; -- let the compiler figure it out!
begin
     return result;
end address_to_string;
```

Figure A-3. Example of String Conversion Function

There was some question whether this approach (that uses Chapter 13 features from the Ada Language Reference Manual) would work on all compilers. For example, at one point the DECAda compiler would not accept such specifications.

Another issue is the general question of whether it is better to be passing around an Ada `access` pointer to arrays or to be passing around the actual Ada arrays themselves. For example, the Xlib `quark` entity is passed around as a `quark_list` (an access pointer) instead of as the base `quark_array` (which is an array). From the C mindset, using `quark_list` makes more sense because treating values as pointers to the start of arrays is what the underlying C/Xlib implementation does. But to Ada applications programmers, being able to

deal with the `quark_array` directly makes more sense. Ada has rather extensive language support for dealing with arrays, why not let the application programmer always deal with them directly?

POSIX P1003.5 is already handling strings between C and Ada. P1003.5 was forced to develop their own custom `Posix.Posix_Strings` type and associated operations (forming an entire `Posix_Strings` ADT) because they (by definition) have to deal with the operating system directly. The Ada/Xlib binding should eventually have the operating system calls abstracted away; so use of Ada's standard strings was used to make the application programmer's life easier.

A.13 MOVE WINDOW_ID FIELD OF ADA/X EVENT STRUCTURE TO THE NON-VARIANT PART

This change removed the need for extra non-Xlib functionality. This happens to be a complaint that Ada/Xlib binding users have had since the STARS Foundations binding originally came out.

A.14 INCLUDE A TRANSITION FROM X11R3 TO X11R4 WITH CONVERGENCE OF ADA/XLIB

The binding developers examined the changes from X11R3 to X11R4 and integrated them into the Ada/Xlib binding. They also provided alternate `make` scripts of dummy C functions to allow building and execution of the bindings with X11R3. The rationale was that many users require X11R4 bindings for their work and adding the additional interfaces now will save the convergence from extra work in the future.

There are already some low-level C functions in both the SAIC and Unisys bindings for dealing with the bit-level boolean and mask operations. The solution to this issue would just add some dummy X11R4 C/Xlib procedures to this existing C library, so any X11R3 users would not get complaints about undefined functions from their Ada linker.

A.15 SPARSE TYPE RANGES

There were some other quirky Xlib type definitions that were worked out. Originally the definition of an angle was defined as:

```
type angle is range 0..359;                -- you'd think this was obvious
```

but an existing STARS Foundation's Ada/Xlib binding user pointed out to us that Xlib has some odd assumptions on how to represent fractional angles, so the correct type definition is:

```
type angle is range 0..(360*64)-1;
```

Another quirk in Xlib is the definition of key codes, which are nominally defined as being between 8 and 255. Unfortunately, a type definition of:

```
type key_code is range 8..255;          -- this won't work!
```

is incorrect because the C/Xlib code Ada binds to allows the use of 0 as a special key code indication (not unlike the earlier example of `null_address`). So the corrected definition is:

```
type key_code is range 0..255;
```

A more rigorous Ada specification notation – such as Stanford University's Annotated Ada (Anna) (Luckham 87) – could have been used to document this noncontinuous type/value assumption (where the values `1..7` are really not supposed to be used). The type would look like:

```
type key_code is range 0..255;
--| ∀ K: key_code ⇒ K=0 or K≥8;        -- Anna annotation
```

A.16 RECTANGLE AND POINT PARAMETERS

The binding developers decided that making the changes to Ada/Xlib to eliminate *all* the references to the non-C/Xlib types `rectangle` and `point` was unnecessary. There were over 70 Ada procedures that used these types; in some cases the parameters are legitimate references to rectangle and points instead of convenient bundlings of loosely-related parameters. SAIC and Unisys changed only those Ada/Xlib procedures that really do not need to use these types.

A.17 OTHER ADA/XLIB ISSUES

One philosophical (or religious) issue that applies to Ada/Xlib as well as Ada/Xt was whether to pass subprogram arguments as `in out` or `in` mode parameters when those arguments are access pointers. In some cases the subprogram's body will not change the (access) pointer to the object being pointed to (so `in` mode would be legal), but the internal value of the object being pointed to will change (where the access pointer is followed and to memory pointed to is modified). The binding developers decided as a matter of politeness to always pass such parameters as `in out` so the subprogram's caller realizes that the object might change.

The `X_Address` data type is defined a subtype of `system.address` instead of a private type. This would allow the definition of `Zero_X_Address` in Ada/Xlib to be `X_Address(0)`. This will eliminate the need for Ada/Xt programmers from having to write many conversion functions (to get an address of 0) since they will be able to use the definition of Ada/Xlib's configuration dependent value.

To satisfy the linker, the routine `XWidthDrawWindow` was renamed `XWithdrawWindow` in the source file `R4_stubs.c` (yes, the dangers of those case-sensitive languages). The routines

XrmDestroyDatabase and XrmDatabase were also added for X11R3 compatibility. The files or.c and and.c were eliminated, with their routines placed in mask.c. The pragma Interface definitions for "_or" and "_and" were removed from the X_Lib_Interface package because they were causing linker problems.

Because of the dope vectors used on some Ada compilers, the arrays screen_list and window_list have to be fixed before any access is made to them (particularly screen_list[0]) so one gets the right numbers. There was concern that some VAX/VMS compilers, such as the new Karlsruhe Ada compiler, would have problems with big endian versus little endian data.

APPENDIX B

DETAILED ADA/XT CHANGES

B.1 ADA PROCEDURE POINTERS

One of the issues that the STARS binding developers resolved was how to construct an abstract Ada callback construct that could transparently be used to encapsulate both SAIC's implementation approach (using task type pointers) and Unisys' approach (using the 'ADDRESS attribute of a subprogram). A code extract outlining this callback is shown below:

```
package subprogram_pointer is
    type func_ptr is private;
    null_func_ptr : constant func_ptr;

    procedure execute (the_obj : in func_ptr;
                       widget_id : in widget;
                       closure   : in x_address;
                       call_data : in x_address);

    function execute (the_obj :  in func_ptr;
                      widget_id  :  in widget;
                      closure    :  in x_address;
                      call_data  :  in x_address) return boolean;

    generic
      type local_ptr is private;
      proc_id : in out Local_Ptr;
      with procedure user_specified (widget_id :   in out widget;
                                     closure    : in x_address;
                                     call_data  : in x_address);
    package proc_ptr is
    end proc_ptr;                    -- no callable subprograms!
private
    type func;                       -- deferred until package body
    type func_ptr is access func;
    null_func_ptr : constant func_ptr := null;
end subprogram_pointer;
```

Figure B-1. Ada/Xt Subprogram Pointer Template

This program extract uses a fairly common Ada data type information hiding technique, where one defines a data type (`func_ptr`) that is an access to another type (`func`), which is declared in the private part of the (`Subprogram_Pointer`) package and does not have to be fully defined until the package's body is written. In SAIC's implementation, `func` will be implemented as an Ada task type. In Unisys' implementation, `func` will be one of their Procedure Control Blocks (PCB) that will hold the `subprogram'ADDRESS` of the Ada subprogram to call back.

The `execute` procedure is what calls either the `handle` task entry point of SAIC's approach or works through the `subprogram'ADDRESS` of Unisys' approach (depending on the procedure's implementation in the `Subprogram_Pointer` package body). The `execute` function is used for those Ada/Xt callbacks that return a value. Note that the `execute` subprograms have four arguments in contrast to the usual X convention of having just three arguments in the callbacks (the `widget_id`, `closure`, and `call_data`). The fourth argument (`the_obj` that holds the callback) should not be that confusing to widget developers using the C/Xt coding styles and documentation.

The generic `proc_ptr` package represents a compromise between Ada's private type visibility rules, the need for passing user-defined procedures around, and a desire to limit the damage caused by incorrect widget specifications. In addition to encapsulating the different SAIC and Unisys approaches from the widget developer's point of view, they also want to encapsulate the whole Ada/X widget callback mechanisms from the Ada application developer's point of view. The application developer will pass the local application subprogram to call in response to some X activity through an instantiation of the `proc_ptr` package with that subprogram as the `User_Supplied` generic formal subprogram parameter. The arguments to the `User_Supplied` subprogram are the standard ones for Xt intrinsics, so application developers will not notice much of a change from the C/Xt style.

The `local_ptr` type and `proc_id` object are used to pass in the (private) `func_ptr` type defined earlier and the instance of that type that will be managing the callback. Because the application developer as well as the widget developer is allowed to modify the (Ada) subprogram that is called back in response to some Xt activity, everyone needs **`in out`** access to the callback object. The `proc_ptr` package is a convenient way to bind all these related data types, objects, and subprograms together so they can have at least some compile-time checking and still be modifiable during the execution of an application. The binding takes place at run-time when the `proc_ptr` package is elaborated and statements in the elaboration code within the package's body can set up various pointers and objects. Note that there are no visible (exported) subprograms or types within the specification of `proc_ptr`; an actual call to the `User_Supplied` procedure is made through passing the `proc_id` callback object and the three arguments of `User_Supplied` to the `execute` subprogram.

Also note that both the `local_ptr` and `func_ptr` types are visible, and yet appear to be duplicates of each other. The binding developers decided to do this because they did not want to force implementation details (either tasks or generics) of Ada/Xt into each callback's body, which would complicate the widget definitions.

Some test programs were written and tried on all of the Ada compilers to which the STARS binding developers had access. They found that this callback approach works with either the

SAIC task type or the Unisys `subprogram'ADDRESS` approach. One question was, what would happen if either the widget writer or the application developer passes arguments of the wrong type into the instantiations of the generic `proc_ptr` package? Since private types are used as the formal arguments, what happens when someone is foolish enough to instantiate with an integer type instead of the `func_ptr` type? Obviously things would blow up, but would they do so at compile or at run-time? Several programmers have noted that one problem with existing C/Xt intrinsics is that one does not learn of these subtle errors until run-time (when the user gets a UNIX `core` dump file). After some experimentation, the STARS binding developers decided that all the Ada compilers that were tried would detect (at compile-time) major instantiation errors. There remain more subtle errors (such as where the subprogram's profile matches what the `User_Supplied` subprogram expects, but the semantics are wrong) that the Ada compiler will not detect. Here the Ada/Xt implementors (SAIC and Unisys) and any future Ada/X widget developers will have to be very careful in documenting the assumptions on these callback subprograms. Perhaps careful English, Anna, Library Interconnect Language [LIL (Latour, 1990)], or other documentation aids would have to be provided.

One question is how to add an `inherit` procedure and/or function (placed immediately below the `execute` ones) in the `Subprogram_Pointer` package. They implemented an `inherit` subprogram to handle all the work needed to pass information through *both* inheritance paths. Ada/Xt will have two potential inheritance hierarchies. One is the subclassing hierarchy, largely defined by the widget developers, where a child widget class will inherit information from its parent class. The second is the hierarchy defined largely by the application developer, where (enclosing) widgets can inherit information (such as run-time user interaction events). As with the assumptions on the `User_Supplied` subprogram described above, the widget developers and application developers will have to be very careful in documenting the use of this `inherit` subprogram.

Within the body of `Subprogram_Pointer`, the SAIC implementation would set up a task type pointer and Unisys would set up another PCB to the `inherit` subprograms. One question that this approach raises is whether all these pointers and objects will be elaborated before they are used. The STARS binding developers decided that the widget developers will have to insert the appropriate `pragma Elaborate` calls in their source code so the application developers can be sure everything is set up as they define and use widget callbacks. There was some concern that objects of type `func_ptr` (such as `proc_id`) would not be initialized to their values before they are used as elements in a record (e.g., in top-level widgets like a `shell`). Since these objects do not get initialized until the statements of the `proc_ptr` package body are executed, the object may not be fully elaborated yet.

Declaring `inherit` as an explicit subprogram will add some call overhead, but it does insure that objects are elaborated and visible to the outside world. An explicit call to `inherit` will force the elaboration of the `proc_ptr` package and so cause the creation of the proper `proc_id` object. Naturally the application developer should not abuse this visible `inherit` subprogram either.

The `Subprogram_Pointer` package described earlier in figure B-1 is really a template for the actual Ada/Xt intrinsics packages that will be in Ada/X. SAIC and Unisys have identified 34 major callback packages that correspond to the major base widget types (such as the shells),

where the parameter profiles and function return values (if any) will vary slightly between callbacks. Unisys has defined a UNIX `awk` script that generates these callback packages from a template file and the appropriate arguments. These `awk` scripts could be modified to generate the different implementation calls (SAIC's task entry points or Unisys' PCBs) as needed. SAIC and Unisys have held some discussions with several European Ada compiler vendors (Systeam and the University of York) to see if the coming generation of Ada compilers can make this even easier. Note that these `Subprogram_Pointer` gymnastics used by the widget developers are not visible to the application developers, who will continue to use the visible subprograms to define the callbacks.

One application design level question is how best to represent this `Subprogram_Pointer` paradigm within the design of an Ada application. Figures 4-3 through 4-6 used a Buhr-style notation to show the interconnections between Ada units in general and Ada/X widgets in particular. Ada application developers may want to use similar high-level graphics to show the interactions between their custom-developed units and those reused from Ada/X. An important part of using these high-level graphical notations is showing the callback connections and interactions between the application and Ada/Xt. But as shown in figure B-1 with `Subprogram_Pointer`, the physical source code connection (involving generic package instantiations, private objects, etc.) are much more complicated than the more abstract connection semantics of Xt callbacks. Application developers may want simple (perhaps provided canonical forms) abstract representations of the Ada/Xt package(s) design, so the important concepts of their application's design does not get lost in the details of the coding.

B.2 WIDGET RECORDS

The widgets developed by SAIC and Unisys (especially the common base widgets, such as a `shell`) are different layouts for the fields of data within the Ada records that encapsulate each widget. If a common Ada/Xt intrinsics specification allows common widgets to be built on top of it, then common layouts of base widgets have to be used.

A further widget record issue has to do with how early users of the Ada/Xt work would be able to transition their widgets from the current implementation choices of a base widget set (currently the Athena widgets) to whatever widget set(s) STARS and other organizations eventually decide to use (such as Motif and/or OL). SAIC and Unisys are assuming that Ada application developers can start using the Athena widgets and then cut over to Motif (or something else) when that becomes supported within Ada/Xt. Given the differences between widget records and class hierarchies, a cut over effort may be more difficult than the binding developers think.

One issue that is related to Ada/Xt widget record layouts are the convenience functions that commercial widget sets (such as Motif) use to give application developers easier access to Xt intrinsics information. If Ada/Xt work is to support a commercial widget set, will all the convenience functions have to be programmed in addition to standard Xt functions?

An issue with the current C/Xt intrinsics implementations is that when a widget instance is created from it parent class, any placeholder information (such as character strings that refer to some other named entity) are compiled into pointers to the actual entity location. With

C/Xt, this compilation is done in place, so the new pointer information overwrites the old data in the C structure. Neither SAIC nor Unisys compiles in place the Ada records that encapsulate widgets; instead a new record is used. This is a much cleaner way of creating widgets and filling out their record information, cutting down on abuse of `system.address` and memory leakage.

SAIC and Unisys had used fairly compatible names in their two implementations, so coming up with a common specification was not that hard. There was some cleanup of remaining uses of the now-banished (or encapsulated) `system.address` data type. The alignment of record components on 32 bit boundaries for clean heap usage during RM usage was defined. They also set up boolean data types that have 8 or 16 bits, as needed. The goal is to allow widget developers to share whatever widgets are developed on top of either SAIC's or Unisys' Ada/Xt implementation. The question of whether or how to support a commercial widget set (such as Motif's or OL's) remains open.

One discussion was about what needs to be done if Ada/X has to use existing widgets. If existing C/Xt widgets are to be (re)used, then these Ada/Xt record layouts may need an extra field to handle the compiled and uncompiled forms of resource (from the RM). C/Xt addresses this by having only one field that initially points to the uncompiled version. When compiled, the new binary data overwrites in place the old data so the same record field can point to either version. Since Ada's strong typing model does not allow one to cheat like this, extra fields may have to be added to the records to hold the compiled and uncompiled versions. This changes the length and/or the order of the record components, making them incompatible with the existing C/Xt widgets and defeating interoperability.

There are some drastic hacks binding developers could employ to try to reuse other record components to point to the compiled and uncompiled versions of resources. For example, the `size` field isn't used in some cases so size could be used to point to a compiled resource. But using `size` as resource pointer is hardly intuitive from the application developer's point of view; this hack could cause more confusion than solving the real problem. Some pointed to this problem as another reason not to try to integrate existing C/Xt widgets in favor of all Ada widgets.

There are other problems associated with C/Xt widget reuse as well. For example, an Ada program would have to go through a C/Xt widget's components to make sure that values such as `null` are correct as far as Ada's concerned (not all Ada compilers assume 0 = `null`). If such an Ada application made that change, would it be obligated later to change the widget back to what C/Xt expects?

A question related to widget record layouts was whether the `shell` widgets can be created by calls to either the `Xt_App_Create_Shell` or `Xt_Create_Managed_Widget` subprograms. This problem eventually went away when the binding developers determined that `Xt_App_Create_Shell` was defined only for X11R2 backward compatibility, and so the X11R3-based Ada/Xt work did not have to support it.

The binding developers discussed whether the (compiled) resources should be done using untyped byte arrays and how to allocate such arrays. One could create such an array, throw the data into it, and then convert it to the appropriate widget class record. There was concern

that these arrays would introduce array dope vectors with some Ada compilers that would throw all our elaborate record length calculations off. Some wanted to use arrays instead of records to handle cases such as `Change_Geometry`, where local copies of widgets are made and then copied to their destination. This work is done in the `Xt_Intrinsics` package, which does not have visibility into the current widget's data structure (see figure 4-10), and so will not know how long the widget record is. C/Xt addresses this by passing a `length` field so the intrinsics can use a byte copy on whatever is in the widget record structure. The Ada/Xt binding developers want to use a similar scheme for Ada.

An alternative discussed again was to find some unused (or little-used) field in the record and reuse it as a pointer to an Ada creation function that returns an instance of the widget record. Note that this would not be carefully controlled like the callbacks described earlier; this would instead be a simple C-style subprogram pointer hack. Some felt we should try this because of the inconsistency problems discussed earlier and the dangers associated with blindly passing Ada `subprogram'ADDRESSes` around.

Another area for discussion was just how to represent such untyped data. Some advocated arrays of bytes, others wanted to use the existing `string` data type. A byte array would involve a call to UNIX's `malloc` function, while `strings` could use existing Ada creation and destruction functionality. Untyped Ada data makes some nervous; they lean toward typing the widget record data more and worrying less about C/Xt compatibility. An alternative discussed was getting the consortia (OSF and UI) to add another field (unused by them) that Ada/Xt could use for its purposes. Other language bindings to Xt could make good use of this field as well.

The binding developers discussed what should be returned to a caller when a call is made to create a widget. Currently, Unisys returns a `system.address` while SAIC returns an access to the appropriate widget. Eventually, they decided to attempt a more abstract return data type, with the implementation details hidden. This looks somewhat similar to the abstract `Subprogram_Pointer` callbacks described earlier.

With some of these higher level issues out of the way, they worked out the specifics of how the different Ada/Xt widget record layouts are done. These are given in the code samples in table B-1:

Table B-1. Widget Record Layouts

```
type Core_Part is record
    self :                  widget;
    widgetclass :           widget_class;
    parent :                widget;
    xrmname :               Xrm_Name;
    being_destroyed :       Xt_Boolean;
    destroy_callbacks :     Xt_Callback_List;   -- starts as an
                                                -- array, compiled to linked list
    constraints :           X_Address;          -- it's really integer data
    x, y, width, height :   dimension;
    managed :               Xt_Boolean;
    sensitive :             Xt_Boolean;
    ancestor_sensitive :    Xt_Boolean;
    event_table :           Xt_Event_Table;
    tm :                    Tm_Record;
    accelerators :          Xt_Translations;
    border_pixel :          pixel;
    border_pixmap :         pixmap;
    popup_list :            widget_list;
    num_popups :            cardinal;     -- used with Unisys, for C/Xt
    name :                  string_pointer;
    screen_id :             screen;
    colormap :              color_map;
    window_id :             window;
    depth :                 depth_type;   -- possibly cardinal instead
    background_pixel :      pixel;
    background_pixmap :     pixmap;
    visible :               Xt_Boolean;
    mapped_when_managed :   Xt_Boolean;
    pad : 16 bits;                        -- to be sure widget ends on 32
end record;                               -- bit boundary (with Booleans)

type core_class_part is record
    superclass :            widget_class;
    class_name :            string_pointer;
    widget_size :           cardinal;
    class_initialize :      Xt_Proc;
    class_part_initialized :Xt_Widget_Class_Proc;
```

```
    class_inited :          Xt_Boolean;
    initialize :            Xt_Init_Proc;
    initialize_hook :       Xt_Init_Proc;
    realize :               Xt_Realize_Proc;
    actions :               Xt_Actions_List     -- could be compiled
                                                -- and result in date overwritten
    num_actions :           cardinal;
    resources :             resource_list;      -- also could be compiled
    num_resources :         cardinal;
    xrmclass :              Xrm_Class;
    compress_motion :       Xt_Boolean;
    compress_exposure :     Xt_Boolean;
    compress_interleave :   Xt_Boolean;
    visible_interest :      Xt_Boolean;
    destroy :               Xt_Widget_Proc;
    resize :                Xt_Widget_Proc;
    exposure :              Xt_Expose_Proc;
    set_values :            Xt_Set_Values_Func;
    set_values_almost :     Xt_Almost_Proc;
    get_values_hook :       Xt_Args_Proc;
    set_values_hook :       Xt_Args_Func;
    accept_focus :          Xt_Accept_Focus_Proc;
    version :               Xt_Version_Type;
    callbacks_private :     Xt_Offset_List_Ptrs;
    tm_data :               Xt_Tm_Data;
    query_geometry :        Xt_Geometry_Handler;
    display_accelerator :   Xt_String_Proc;
    extension :             X_Address;
end record;

type composite_rec is record
    children :              widget_list;
    num_children :          cardinal;
    num_slots :             cardinal;
    insert_position :       Xt_Order_Proc;
end record;

type composite_class is record
    geometry_manager :      Xt_Geometry_Handle;
    change_managed :        Xt_Widget_Proc;
    insert_child :          Xt_Widget_Proc;
    delete_child :          Xt_Widget_Proc;
    extension :             X_Address;
end record;
```

```
type constraint_widget_rec is record
    empty : X_Address;                          -- dummy placeholder
end record;                             -- may get rid of this empty record

type constraint_class_part is record
    resources :             Xt_Resource_List;
                                        -- could get compiled over itself
    num_resources :         cardinal;
    constraint_size :       cardinal;
    initialize :            Xt_Init_Proc;
    destroy :               Xt_Widget_Proc;
    set_values :            Xt_Set_Value_Func;
    extension :             X_Address;
end record;

type object_widget_part is record
    self :                  widget;
    widgetclass :           widget_class;
    parent :                widget;
    xrmname :               Xrm_Name;
    being_destroyed :       Xt_Boolean;
    destroy_callbacks :     Xt_Callback_List;   -- starts as an
                                -- array, compiled into linked list
    constraints :           X_Address;          -- it's really integer data
end record;
```

The related `object_class_part` record has the identical fields of the earlier `object_class_part` record type. The remaining widget records (such as those for the `shells`) had similar layouts and record definitions (except for the occasional ordering of a pair of components).

B.3 WIDGET CREATION

When Xt widgets are created, some computer memory must be dynamically allocated. Naturally that means the memory should be reclaimed at some later point to prevent memory leakage or loss. If all the Ada/Xt widgets are written entirely in Ada, then they can manage their own memory. If Ada/Xt interfaces or reuses C widgets, then there would have to be explicit calls to the C/Xt widget's `create()` entry point. The binding developers identified some alternatives for creating widget memory for Ada:

- explicit calls to C `malloc()`, which is what C widget reuse would require,
- a new `byte` function within Ada that does the same thing from the Ada side,
- place a `create` function in each widget package, and
- use a generic `create` package that's instantiated for each widget.

Note that the first and second alternative above will do the creation work in the intrinsics, while the third and forth do the creation within each widget.

B.4 INTRINSIC WIDGET PACKAGING

In Ada/Xlib, there is a giant package `X_Lib` that is withed into any Ada applications program that wishes to make any Xlib calls. For Ada/Xt, there will be a corresponding package `Xt_Intrinsics` that application developers will need to with in to make Xt calls. In addition, applications will have to "with" in the packages that define various base widget classes such as `shell` and `object`. SAIC and Unisys initially had different assumptions about how different packages (such as `Xt_Core`) are withed in. Straightening this out required looking at how widget subclassing was done so new widgets (and their packages) are visible.

A related part of packaging is how to document all the methods and other class information provided in the Xt intrinsics and the widgets built from it. Unisys made a start at textually defining in the package headers how method information is visible (and inherited) to applications. As described above, documenting the relationships between Ada execution and Xt inheritance behavior is nontrivial.

A naming convention was established that all the Ada/Xt identifiers would follow. As with Ada/Xlib, underscore separators will be used between words in an identifier (e.g., `Xt_Seperate_With_Underscores`). As shown in the `Subprogram_Pointer` package earlier, the (public) specification part of Ada/Xt will hide the use of `System.address` from the application developer. As shown earlier, either an opaque type (such as `X_Address`) is used or the data type/object uses a more descriptive name.

As discussed earlier, C/Xt uses different data structures with the same name (differentiated only by a '_' prefix character). The binding developers decided to use Ada's name overloading, so the same (C/Xt) identifier name can be used for both the '_' prefixed case (used by widget developers) and thenonprefixed case (used by application developers). As part of the overall Xt intrinsics packaging issue, they agreed to follow Unisys' idea of using semiprivate packages to encapsulate those types and operations used only by the widget developers. This means the overloaded Ada types, of interest mainly to the widget developer, will be separated from the types used by the application developers anyway. As with the callback and inheritance mechanisms defined earlier, careful documentation will have to be provided so application developers do not accidently (there being no way to prevent deliberate misuse) use the wrong data types.

One packaging issue was how to handle the inconsistent naming conventions used at various points in the C/Xt binding. Unisys tried to support multiple names for the same entity by

defining a Renamed_Xlib_Types package that a programmer could with, and then use to support both C/Xt naming styles. After further checking, it was decided this Renamed_Xlib_Types package was an unneeded duplication of an already crowded name-space. The duplicate C/Xt names were either eliminated or moved to the appropriate Ada/Xlib or Ada/Xt package.

An overall intrinsics packaging question was how to place the various Ada/Xt subprograms into the right packages so there would be a minimum amount of duplication, and yet not force the widget developers and application developers to have to with in large numbers of Ada packages. Coming up with a common intrinsics packaging scheme was complicated because SAIC and Unisys had different conventions on what had to be withed in. Since C/Xt did not define a packaging scheme, often SAIC and Unisys took different choices of which Ada packages into which to place a given subprogram. Even when the same general packages were used; there were cases where SAIC used a _management suffix on some package names where Unisys did not. Unisys had their own Command_Line_Arguments package that allowed access to the original (UNIX) argc and argv arguments before the X-specific (as with geometries) arguments were filtered out.

As with Ada/Xlib, there was an argument whether the Ada/Xt functionality should be encapsulated into several Ada library packages or grouped together into one giant package. As with Ada/Xlib, the argument for using multiple library packages so the name space is somewhat limited and the application program has some choice as to what is withed in a program was lost. Instead there will be a giant Xt_Intrinics package that will contain subpackages that group the major Ada/Xt types and subprograms together. As with Ada/Xlib, an argument was made that the Ada/Xt entities were so mutually intertwined that it was impossible to figure out a clean way of separating them into different library packages. Both the widget developers and application developers will need only one "with Xt_Intrinics;" statement to make all these entities (and subpackages) available.

The original Ada/Xt work used an Xt_Ancillary_Types package that held a loose collection of various data types. The STARS binding developers decided to eliminate this package in favor of moving all the data types up to the specification of Xt_Intrinics. This makes data types such as Xt_Boolean visible to all the lower level subpackages. An Xt_Boolean type is used instead of the standard Ada boolean type because compatibility with (Ada/Xlib) layouts and different compiler representations require that the (Xt_Boolean)'WIDTH attribute be settible. To avoid having each programmer create own objects of this type, Xt_True and Xt_False objects are defined so that programmers can take the 'ADDRESS of.

As with Ada/Xlib, the widget developers and application developers should not have to define their own versions of Unchecked_Deallocation for all the objects of Ada/Xt data types that they create. Overloaded Xt_Free procedures are defined to encapsulate this.

Another long discussion was over where to place the 34 instances (such as those created with awk scripts) of the Subprogram_Pointer package used for the callbacks. Initially, all of these callback packages are encapsulated in an Xt_Procedure_Types (sub)package so all these packages are grouped together. As explained earlier, the widget developer and application developer may have to deal (at the specification level) with the proc_ptr generic package that is within Subprogram_Pointer. So proc_ptr is a subpackage of

Subprogram_Pointer, which is a subpackage of Xt_Procedure_Types, which is a subpackage of Xt_Intrinics. There is concern that sub-sub-sub-packages are just a little too complicated for an average Ada programmer to deal with. Some thought the Xt_Procedure_Types package should be eliminated, with the Subprogram_Pointer instances moved up to the top level of the Xt_Intrinics package specification.

The TIM attendees decided to use the Unisys approach of semi-private packages (sub-subpackages within the overall Xt_Intrinics package being used) to encapsulate those Ada/Xt entities within an Xt_Intrinics subpackage that is of interest only to a widget writer. Note that it is these semi-private packages that introduce many of the complex interrelationships among the Xt_Intrinics subpackages.

After a lengthy discussion of what goes where, the STARS binding developers came up with a preliminary description of the Xt_Intrinics subpackages, and the Ada/Xt subprograms that would be placed in each of them. Note that the list below does not show all the semi-private packages; one can assume where C/Xt defined them, then Ada/Xt will have them. There were some instances of documented semi-private entities that were not used anywhere; these will still be in Ada/Xt since C/Xt's specification defines them as well, and future widget developers might use them. These will be placed in the (sub-sub)package bodies instead of the specifications until someone can determine if anyone really uses them. Someone could look at the Motif and OL source code to see if these obscure routines are ever used, but there was some concern about violating various copyright agreements if that same person then went on to write widgets (perhaps influenced by the source code they saw). Note that there are many subprograms that have been documented as being for X11R2 backward compatibility-only that were not implemented (such as all those non-app routines that used the now-discouraged default application context instead of the user defined one). The allocation to packages is shown in table B-2 below.

Table B-2. Allocation of Subprograms to Packages

Package Name	Subprograms
Xt_Resources	all the conversion subprograms, which do not have to be enumerated here,
Xt_Event_Management	Xt_Add_Input, Xt_Remove_Input,Xt_Add_Timeout, Xt_Remove_Timeout, Xt_Add_Grab, Xt_Remove_Grab, Xt_Set_Keyboard_Focus, Xt_Call_Accept_Focus, Xt_Pending, Xt_Peek_Event, Xt_Next_Event, Xt_Process_Event, Xt_Dispatch_Event, Xt_Main_Loop, Xt_Add_Work_Proc, Xt_Remove_Work_Proc, Xt_Add_Event_Handler, Xt_Remove_Event_Handler, Xt_Add_Raw_Event_Handler, Xt_Remove_Raw_Event_Handler,

	Xt_Build_Event_Mask, Xt_Add_Exposure_To_Region*, Xt_Window_To_Widget*, Add_Forwarding_Handler*, Get_Window_From_Event*†, Xt_Make_Toolkit_Asynch§, Xt_Set_Asynch_Event_Handler§, Xt_Timer_Callback,
Xt_Translation_Management	Xt_Add_Actions, Xt_Parse_Translation_Table, Xt_Argument_Translations, Xt_Override_Translations, Xt_Uninstall_Translate, Xt_Parse_Accelerator_Table, Xt_Install_Accelerators, Xt_Install_All_Accelerators, Xt_Set_Key_Translator, Xt_Translate_Keycode, Xt_Case_Cvt, Xt_Register_Case_Cvt, Xt_Convert_Case, Xt_Initialize_State_Table, Get_TM_Trans_Offset†, Compile_Action_Table†, Compile_Action_List†,
Xt_Geometry_Management	Xt_Make_Geometry_Request, Xt_Query_Geometry, Xt_Make_Resize_Request, Xt_Move_Widget, Xt_Resize_Widget, Xt_Configure_Widget, Xt_Translate_Coords,
Xt_Selection_Management	Xt_Set_Selection_Timeout△, Xt_Get_Selection_Timeout△, Xt_Get_Selector_Value, Xt_Get_Selector_Values, Xt_Own_Selection, Xt_Disown_Selection,
Xt_Popups	Xt_Create_Popup_Shell, Xt_Popup, Xt_Popdown, Callback_Procs†, Menu_Popup△, Menu_Popdown△,
Xt_Callbacks	Xt_Add_Callback, Xt_Add_Callbacks, Xt_Remove_Callback, Xt_Remove_Callbacks, Xt_Remove_All_Callbacks§, Xt_Call_Callbacks, Xt_Has_Callbacks, Get_Callback_List†,

* may be moved to the Xt_Utilities package instead of here.

† note no Xt_prefix.

§ not in X11 documents.

△ could be X11R2 only.

	Xt_Add_Callback, Xt_Call_Callbacks, Xt_Remove_All_Callbacks, Xt_Free_Callback_List, Xt_Compile_Callback_List, Xt_Get_Callback_List, Xt_Get_Uncompiled_Callback_List§,
Xt_Instance_Management	Xt_Get_GC, Xt_Set_GC, Xt_Destroy_GC, Xt_Toolkit_Initialize, Xt_Create_Application_Context, Xt_Destroy_Application_Context, Xt_Display_Initialize, Xt_Open_Display, Xt_Close_Display, Xt_App_Create_Shell, Xt_Create_Widget, Xt_Destroy_Widget, Xt_Create_Managed_Widget, Xt_Set_Mapped_When_Managed, Xt_Map_Widget, Xt_Unmap_Widget, Xt_Realize_Widget, Xt_Unrealize_Widget, Xt_Create_Window, Xt_Manage_Child, Xt_Manage_Children, Xt_Unmanage_Child, Xt_Unmanage_Children,
Xt_Error_Management	Xt_Error_Msg, Xt_Warning_Msg, Xt_Error, Xt_Warning, Xt_Set_Default_Error_Handlers,
Xt_Utilities	Xt_Name_To_Widget, Xt_Widget_To_Application_Context, Set_Ancestor_Sensitive†, Xt_Is_Sensitive, Xt_Set_Sensitive, Xt_Is_Realized, Xt_Is_Composite, Xt_Display, Xt_Screen, Xt_Window, Xt_Parent, Xt_Is_Managed, Xt_Class, Xt_Superclass, Xt_Is_Subclass, Xt_Check_Subclass, and Xt_Is_Shell.

Note that many of the subprograms above that do not have the Xt_ prefix are used as local Ada/Xt convenience functions (such as providing offsets for RM) instead of implementing some behavior specified for C/Xt as well.

In the current Xt_Resources (sub)packages, there are a collection of subprograms (and their arguments) that pass a _pointer around that can either be the address and length of the array to fill up or a null (this a a common C programming paradigm). Binding developers would like to eliminate having to pass the length parameter around, since Ada allows the subprogram being called to determine the length through the 'LENGTH attribute. Given the way the Xt_Resources arrays are defined, some wondered if empty arrays that are declared by "array a[1..0];" would still return a 'LENGTH attribute of zero. After some checking, the developers decided that all the Ada compilers they had access to (which *doesn't* include Meridian or other compiler vendors who use dope vectors) would correctly return zero in this case. So they could eliminate all those extra length parameters.

SAIC and Unisys have very different approaches on how to implement RM, so the binding developers came up with a common specification (with different bodies) for the `Xt_Resources` package. Even with the same specification, there are subtle implementation differences that a user will have to be aware of. For example, the predefined resources are compiled (in the RM sense, not the Ada source code compiler sense) differently between them. Unisys does this at package elaboration time, while SAIC waits until an explicit call is made.

SAIC points out that the current X11R4 RM uses compiled resource caching versus the scheme used in X11R3 (on which what both the Ada/Xt RMs are based). X11R4 has chosen to make use of some of the strange side effects of the internals of this (cached) compilation process. For example, the X11R4 RM will note the starting and ending quark number into a range that is used by later functions to determine which resources were really precompiled. This scheme will eventually break Unisys' approach, which intersperses resource precompilation with all the other activities that take place during elaboration. Therefore, there is no longer a clean range of quarks deep within the RM that can be destroyed and reclaimed at will; some of the quarks used by Unisys may have nothing to do with RM. SAIC believes the current X11R3-based Ada/Xt RM should reflect this change that future X11R4-based Ada/Xt implementation will have to deal with.

Other issues such as the representation specifications for raw and private resources would flow from these issues. There was some concern that the visibility of these representation specifications would prevent widget data types (such as the `Core` widget) from being private data types. It is not clear if the `set_arg` subprogram needs to convert its input arguments as a standard `integer` type or whether a generic instantiation needs to be done for each type (such as was done earlier with the `Subprogram_Pointer` package [template]). These implementation details are deferred to the `Xt_Resources` package body.

B.5 VARIABLE/PROCEDURE NAMING CONVENTIONS

SAIC and Unisys had different naming conventions for the different Ada entities in Ada/Xt (as they did in Ada/Xlib). An example of this is how the C/Xt private information (normally preceded by a '_' character) is represented in Ada (where one cannot begin a name with a '_'). These Ada names could be differentiated from their public cousins (in C/Xt one would drop the'_' for the public name) by fully qualified (sub)package names. A truly ugly solution would be to use `Bar_` as the prefix to those entities whose names are supposed to be private. The best solution was to use Ada overloading; since different data types are being used, the Ada compiler can differentiate through parameter profiles.

B.6 C WIDGET SUPPORT

If SAIC and Unisys were to come up with a standard way of representing widgets, the question came up whether the Ada/Xt intrinsics should be able to handle existing (written in C) widgets. This would be a quick way to exploit (reuse) all the existing C/Xt (commercial)

widget sets being used, making an Ada application developer's job easier. This would also limit the need for STARS to fund a massive widget writing effort.

But there are some problems with attempting a C/Xt widget reuse approach. Ada/Xt would have to know the internal layout of class method (value) information within a widget class and instance. That information is typically provided in the widget link module's symbol table (since there would be no general source code access to a commercial widget set such as Motif or OL). There are known UNIX hacks for extracting this information out of a symbol table; but what if the widget developer has chosen to `strip` (i.e., use the UNIX utility) the symbol table off the load module to save space (a common practice)?

A way around `stripped` widgets is to write a special binder subprogram in C that provides routines into the needed methods, and then use `pragma interface` to allow Ada/Xt to call them. The existing Ada/Xt widgets would have to be adjusted so they exactly match the C/Xt widget layouts. Application developers would then have the choice of using C widgets or developing their own in Ada.

But there are some problems here as well. Application developers could not subclass off of the existing C widgets because the links in the class inheritance chain would be broken at the cross-language boundary. Ada widgets would have to be built from scratch. Losing the ability to subclass and inherit with X widgets would be a major loss of functionality, and would represent a major violation of X's OOD spirit.

Another problem with mixed language widgets are secondary routings into the underlying Xlib layer. Using C widgets means that the underlying C/Xt intrinsics will call their underlying C/Xlib functions directly, bypassing Ada/Xlib. Ada widgets will usually go through Ada/Xt and eventually through Ada/Xlib. Besides the memory allocation and reclamation problems this will cause, the application developer will have to make sure that all the called-back Ada applications functions are elaborated before they are called back to prevent `program_error` from being raised.

This wrapper approach introduces nontechnical problems as well. Applications using Ada/Xt and these pseudo-Ada widgets would have to incorporate and maintain all the C binder code. Unlike the few low-level logical mask operators written in C for Ada/Xlib, this would introduce a lot of C code for the application developers to see (and in some cases, write). Writing this widget binder would require reading the source code of the existing commercial C widgets, so a widget developer knows what the layout of information really is. This could violate the copyrights' and proprietary secrets' clauses of the original writers such as OSF has on Motif.

Deciding whether to support (or not) existing C/Xt widgets requires some tough technical and nontechnical trade-offs. The STARS binding developers lean towards doing everything in Ada as an application maintenance aid, and because reimplementing existing C widgets in Ada is not perceived as being a major effort (done with simplistic brute force programming). Others who have tried direct Ada bindings to commercial widget set implementations (such as Motif) have reported problems with inheritance of widget information.

B.7 INTRINSIC PACKAGING

A packaging issue was how to implement the Xt intrinsics' concept of public and private information within Ada/Xt's specifications and bodies. In C/Xtthepublic definitions of the widgets (the `structs` that a C application developer would [through `#include`] into the source code) are defined in one header (a `*.h`) file. The private definitions of the internal details of the widget are defined in a separate header file (named `*P.h`). This private header file is `#included` only by C programmers who are widget developers and so who need to see the internal details. As described above, normally the private `structs` use a '_' character as a prefix to the `struct's` name.

Ada/Xt also has to package both this public and private intrinsics information so both the Ada application developer and the Ada widget developer can get the data they need. SAIC and Unisys initially took different implementation approaches. SAIC placed each Ada/Xt widget in one Ada package. One source file will contain the package specification (defining all the visible entities) and the other file will contain the package body (where those entities are implemented). The drawback to this approach is that the widget's private definitions (those used only by widget developers, not application developers) are visible to everyone through the common package specification. There is some concern that Ada application developers could access and misuse these private (in the Xt but not in the Ada sense of the word) definitions. To be fair, there is nothing to prevent the determined or foolish C/Xt applications programmer from including and then using the private structs.

Unisys could not find a way of preventing Ada application developers from seeing these private specifications either, but they used one of the awkward bugs of the Ada language and turned it into a feature! Earlier, there were some concerns about Ada/Xlib's use of Ada nested packages within the giant `X_Lib` interface package. Unisys has placed all the private specifications into nested subpackages that are implemented as Ada separate packages. While normally such separate subpackages introduce major visibility and usability problems to a programmer, in this case they help separate the (possibly inexperienced) Ada application developer from the details the (hopefully experienced) Ada widget developer must handle. Unisys calls their separate files semiprivate packages.

Both SAIC and Unisys claim that it would require a major redesign and repackaging effort to prevent an Ada application developer from seeing this semi-private information. One Ada-like argument is to point out that the encapsulation of Xt's conceptual widgets (which are object-oriented in that there is a hierarchy of classes and inheritance of information) really does not map well onto either SAIC's or Unisys' use of Ada packaging. On the other hand, Xt intrinsics really do not map well onto C's even weaker model of packaging through `(#include)` files either. SAIC and Unisys believe that having Ada/Xt follow the same basic C/Xt implementation for packaging allows Ada/X to reuse all the existing C/Xt documentation to define how Ada programmers would use the intrinsics and widgets.

B.8 ADDRESSING

As with Ada/Xlib, there were questions about how to address all the information being created by both C and Ada. SAIC took the approach of trying to avoid the use of `system.address` as much as possible, while Unisys initially used `system.address` throughout their Ada/Xt code. The question is what to use as the opaque pointer to widgets: the values returned by an access allocation or a more conventional `system.address`. Eventually, the binding developers decided to provide higher level interfaces because the Ada application developer should see as little of `system.address` as possible.

B.9 XT RESOURCES

If there was to be a common Ada/Xt intrinsics specification, then that implied that the widget sets that are built on top of Ada/Xt should be shareable across the SAIC and Unisys implementations. Initially that was not the case; SAIC could use Unisys widgets but not the other way around. The problem has to do with how Xt resources were being defined.

Unisys does the compilation of resource information (such as the available widget class/instance slots and their initial values) when the generic Ada package bodies that encapsulate widgets are instantiated. The Unisys Xt RM allows the use of a variety of static class values, such as direct insertions of Ada strings in a class' record. Overridden class values would use dynamic values, such as access string data. User-defined widgets are created from UNIX `awk` scripts that generate the Ada encapsulation package source code from templates.

SAIC encapsulates its widgets within Ada tasks and task types. The class/instances values are more dynamic here, so strings are always being represented through access string data. This means that the current mechanisms for overriding default widget class values were different and incompatible. Eventually, a potential work-around to this problem was identified. SAIC's `create_widget` functions could be encapsulated in a generic package that could be instantiated with the different types of widgets (such as `Xt_Box_Widget`) and widget classes (such as `Xt_Box_Widget_Class`) as needed.

B.10 TREATMENT OF LISTS AND ARRAYS

As with the Ada/Xlib binding, there were variations between how cleanly SAIC and Unisys encapsulated Ada lists and arrays. In some cases, Ada access values are passed around; in other cases they fall back to `system.address` usage. These differences had to be reconciled, much as the Ada/Xlib differences were. Using Ada/Xlib's conventions for operations such as C to Ada string conversions means that Ada/Xt does not have to deal with that interface. There were some problems with Xt's inconsistent use of argument lists that had to be resolved.

B.11 PROPAGATION OF XLIB CHANGES TO XT CODE

The TIMs have established a long list of changes to the underlying Ada/Xlib packages. These changes rippled through the Ada/Xt code that was built on top of Ada/Xlib. Both the common Ada/Xt specifications and the different bodies had to be changed to use the new common Ada/Xlib. Care was exercised so the names of various types were not changed so much that application developers could not look up things in the existing Athena documentation.

B.12 PORTABILITY

Just as Ada/Xlib revealed Ada programming practices that proved to be nonportable to different Ada compilers, the two existing Ada/Xt implementations may have used nonportable or inefficient programming practices that will cause application developers grief when they try to compile and run. These will have to be cleaned up if Ada/Xt is to be considered product quality.

An example of this is SAIC's assumption that an access to a task type can be treated just like a `system.address`. This may not always be true; Ada compilers can do all sorts of weird things with the Task Control Blocks (TCB) that are being pointed to. SAIC also assumes that different task types that have identical entry points with identical parameter profiles can have their access pointers converted from one type to another. This may seem logical, but it depends on the good graces of the internals of Ada tasking run-time executives.

Another assumption shared by SAIC and Unisys is that the run-time executive is smart enough to clean up memory after it is finished with it. For example, SAIC's dynamic task instances are supposed to be cleaned up and have any TCBs that were allocated reclaimed. Not all Ada compilers have followed this theory. Unisys assumes that all the generic instantiations of the callback packages are done efficiently (as with generic body sharing) and reclaimed as needed. But full generic body sharing is not as common as everyone would hope. There is the danger that over the life of a large Ada/Xt application, all the memory will get used up.

B.13 REPRESENTATION SPECIFICATIONS

SAIC and Unisys initially used different conventions for defining the representation specifications used for the various Ada/Xt records. If common Ada/Xt specifications and shared widgets are to be provided, these had to be changes so they were in agreement. Eventually both decided on how to use common definition of bit and byte offsets.

B.14 XT ERROR HANDLING

Initially Unisys had a `Xt_Error` exception defined that was raised when Ada/Xt noticed that input parameters were bad. They also had an `Xt_Exit` exception that could be raised when an application developer wished to terminate the entire program. Since the Ada/Xt internal

intrinsics code will have no handlers for `Xt_Exit` when raised, it will propagate control out of the intrinsics and back to the original application's main program that called the intrinsics' main loop.

SAIC also allowed application developers to raise an exception to break out of Ada/Xt back to their main program; in this case, the application developer would call a designated subprogram that raises this exception. Unisys and SAIC had to be sure they use common exception names and they have common schemes for handling errors. For example, are all the current Xt errors that result in calls to C/Xt's `Xt_error()` function now going to be mapped into one or more Ada exceptions? The STARS binding developers felt it was better to raise an exception to break out of Ada/Xt than to call the UNIX `exit()` routine directly.

B.15 MULTI-THREADED X SERVERS

At some point the X Window System will go to multi-threaded servers as a standard. Already some X vendors have their own private extensions to support multiple threads. Initial reports are that X11R6 will add this. Whenever this comes, an Ada/Xt implementation should (theoretically) be ready since Ada already supports multiple (lightweight) threads in the tasking model.

Currently SAIC's design uses tasks, but all the work being done by the task is done within the rendezvous itself. To be truly multithreaded, the work done by each task would have to be moved to after the rendezvous so both the task and its caller could be running simultaneously. Priorities on these different task (types) would also have to be established. The current SAIC design is tasking-safe since everything is really single-threaded due to work done solely in rendezvous. Modifying either SAIC's or Unisys' design to be multitasking save would take a lot of checking (before any standardization would pass). The emerging POSIX P1003.5 standard could provide ways of encapsulating these issues.

B.16 OTHER ADA/XT INTRINSICS CHANGES

This section contains the list of 117 proposed Ada/Xt intrinsics changes from the initial Ada/Xt bindings and what the STARS binding developers have decided to do about them. This list was prepared by Bill Rosen; any additional comments beyond what Bill stated will be placed after each item.

No.	Proposal	Resolution
1	use `configuration_dependent` clause for arithmetic operations	approved
2	delete **`with`** `Xt_Utilities` clause; not needed in `Xt_Intrinsics` package specification	approved

3	add **with** devices clause for sockets definitions for alternate event input	approved

The devices package (with the definitions of the sockets, file descriptions, and other operating system-dependent data types) would be placed in the devices_.a and devices.a source files. There are other implementation-specific subprograms (such as those dealing with UNIX environment variables), but these are generally hidden within a package's body. Only devices contain generally visible data types. The Xt_Utilities subprograms might eventually be included in some package's body, so the whole Xt_Utilities package might go away.

4	change type modifiers to x_mask_type	approved
5	the Xt_Interval_Id, Xt_Input_Id, and Xt_Work_Proc_Id data types are changed to private data types	approved
6	add type process_context for semi-privates	approved
7	add subtype device for sockets	refer to item #3 above
8	add private types converter_table_rec, converter_table, and Xt_Per_Display for semi-privates package	approved
9	type Xt_Input_Mask moved to event_management	approved
10	Xt_Version_Type changed to cardinal	approved
11	delete the extra Ada exception definitions	approved

The binding developers discussed whether Ada/Xt should define and use additional Ada exceptions such as Xt_Error_Raised, Illegal_Bit_Test, and Xt_List_Overflow in addition to the Xt_Exit exception described in section A-5 above. Since STARS is developing an Ada/Xt implementation instead of a binding (unlike Ada/Xlib), could the toolkit catch and process errors better than C/Xt does (with the oblivion of a UNIX core file dump)? They were not sure C/Xt really separates the fatal errors from simple warnings; there are places in the C/Xt code where execution continues after the error is spotted (no matter what it was).

Ada/Xt could try to go back and characterize every possible error, deciding which can be reported with just a warning message and which also need something like Xt_Error_Raised issued. But they decided this binding does not have the time for that level of thorough analysis. Instead Ada/Xt will issue a warning message and press forward.

The binding developers also discussed whether the whole application should be killed when unrecoverable errors – such as overflowing the resource database – occur, or should warning messages only be output? There is nothing the applications developer can do in this case (besides using fewer resources and/or using default values). Even here they decided that outputing warning messages will have to be enough. Most of these errors should *never* occur. For example, the `Xt_List_Overflow` exception would have occurred only if someone developed a resource and widget class hierarchy such as `resource.class1.class2...class99`, where the built-in limit of 100 in the internal arrays would finally be exceeded. A future reimplementation of Ada/Xt to X11R4 or X11R5 might have the time to do this right since they can go through on a case-by-case basis to see which errors are fatal to an application.

12	delete `Xt_Unspecified_Pixmap`, `max_path_len` and `none` objects, define instead in widget semi-privates or sub-package bodies	examine all to see if they can be removed
13	change `Xt_Version` value to MIT-style version number (version + revision) for checking X11R3 ↔ X11R4 mismatches	approved
14	move `mod_to_keysym_table_rec` and `mod_to_keysym_table` data types to `Xt_Translation_Management` package specification and body	approved
15	change `widget` and `widget_class` to follow Tim Schreyer's proposed widget definition	approved
16	delete `to_widget` and `to_widget_class` conversions; not needed	approved
17	move `Xt_Grab_Record` and `Xt_Grab_Type` to `Event_Management`'s package body	approved
18	move `Xt_Hash_Record` and related declarations to `Event_Management`'s package body	approved
19	move `exclusive_type` to `Event_Management` body	approved
20	move `callback_record` and related declarations to	deferred to check callback package's specification and body `callbackrecord` some more
21	move `Xt_Cw_Clear`, `Xt_Cw_Query_Only`, and `Xt_Sm_Dont_Change` objects to `geometry_management` package	approved

22	change `Xt_Free(Xt_Widget_Geo_Ptr)` to actual function because of deferred widget definition	approved
23	move `callback_struct` to callbacks package spec	refer to issue #20 and make some of these packages private

As mentioned earlier, there is some question whether the 34 packages that encapsulate Ada/Xt callback procedures should be grouped into their own (sub) package under `Xt_Intrinsics`, placed near the intrinsics subpackages they support (such as `error_proc` next to the `error_handler` package), moved to the private part of intrinsics (for those callbacks an external programmer does not use), or some combination of these. Ideally, the callback procedures would be encapsulated with the related ADTs, but there are some callbacks used in many places. Spreading them out could make finding individual packages difficult.

The binding developers eventually decided to treat each of these callback packages on a case-by-case basis. Only those callbacks that are used solely within a subpackage's semiprivate code will be moved into the specification of that package. The others will be placed (alphabetically) together near the front of the `Xt_Intrinsics` package's specification.

24	delete `callback_struct_ptr from` the `Xt_Offset_Record` definition; not in the C or Unisys intrinsics	refer to issue #20
25	move `Xt_Grab_Kind` and `Xt_Popdown_Id` to `popup_management` package	approved
26	change `Xt_Convert_Arg_Record's size` field to type `cardinal`	approved
27	delete `converter_record_` related declarations; these should be in intrinsics package body	approved
28	delete `compiled_translations` declarations; can't find where they are used	approved
29	move `Xt_Action_Record` to `translation_management` package specification and body	approved
30	delete `signature` field of `Xt_Action_Record`	approved
31	delete `Xt_Action_List` and its `Xt_Free`; not used	approved
32	move `Xt_Compiled_Action_Table` to `translation_management` private	approved

33	move `actions_record`, `Xt_Event_Record` to `translation_management` body	approved
34	move `state_record` and `state_pointer` to private and body	approved
35	move `event_obj_record` to `translation_management` body	approved
36	move `tm_kind to translation_management` body	approved
37	move `Xt_Bound_Acc_Action_Rec` and `Xt_Bound_Acc_Action_Rec_Ptr` to `translation_management` body	approved
38	move `Xt_Translate_Op` to `translation_management` semiprivate package specification	approved
39	move `translation_data_record` to `translation_management` body	approved
40	move `Xt_Translations`, `Xt_Accelerators`, and `mod_to_keysym_table` to private part	approved
41	delete `to_translations` function; not sure why it's needed move the following to `translation_management` body:	approved
42	`tm_convert_rec`	approved
43	`late_bindings`	approved
44	`Xt_Match_Procs`	approved (see issue #23)
45	`Xt_Event_Record`	approved
46	`event_seq_record`	approved
47	`tm_event_record`	approved
48	`name_value_record`	approved
49	delete `destroy_list_entry`; not found in C or Unisys	deferred to check for use
50	change `tm_record` to private type	deferred to check some more

The binding developers wanted to check if Unisys' added access functions are used anywhere. They do not want application programmers to access the `state` field, but they do not want to cut off access to needed fields either.

51	change `Xt_Free` and `Xt_Tm_Data` to actual functions because of private `tm_record`	refer to issue #50
52	add `Xt_Compile_Action_Table`; unique to Unisys	approved
53	move `merge_tables` procedure to `translation_management body`	deferred
54	`translation_management` sub-program declaration changes:	
	`Xt_Parse_Translation_Table` (passing an object instead of a pointer)	approved
	`Xt_Translate_Initialize` (move to semiprivate)	approved
	`Xt_Install_Translations` (move to semiprivate)	approved
	`compile_action_table` (change to function and move to semi-private)	approved
	`Xt_Popup_Initialize` (propose to be added)	approved
55	move `Xt_Bound_Actions` type to private remove if `Xt_Action_Proc_List` isn't found or ever used	maybe
56	delete `Xt_Menu_Pop[up, down]_Action` declarations. These should be added to the `app_contexts` action list at `app_context` creation time	approved (see issue #54)
57	add `get_tm_translation_offset` function; unique to Unisys; tentatively move to semiprivate if record becomes private	deferred
58	add `translation_management` semiprivate spec.	approved
59	change `Xt_Event_Table` type to private	approved (see issue #50)
60	add new `Xt_Input_Mask` event type and `null_event_table` object	approved
61	remove `Xt_<no _App>_*` subprograms	approved

62 delete `add_forwarding_handler` procedure; should be in `event_management` body — approved

63 event management subprogram declaration changes (`in` versus `in out` parameters):

`Xt_Dispatch_Event`	approved
`Xt_App_Main_Loop`	approved
`Xt_Free_Event_Table`	deferred
we will consider removal vs. making this semi-private	
`Xt_Add_Exposure_To_Region` (move to `events`?)	approved
`Xt_Window_To_Widget` (move to `events`?)	approved

64 `Xt_Geometry_Management` subprogram declaration changes (use `position` and `dimension` for parameters instead of `pixel`). In subprograms below. They will check `widget_id` parameter mode before committing to these changes. — deferred

```
Xt_Resize_Widget
Xt_Configure_Widget
Xt_Make_Resize_Widget
Xt_Move_Widget
Xt_Translate_Coords
```

65 add `cvt_string_to_boolean_proc` procedure pointer issue object. We need to check if this is used anywhere; otherwise it can move to the body of the conversion package. — deferred (see #63)

66 delete `Xt_Dependencies` procedure; should be in `resource_management` package body — approved

67 resource management subprogram declaration changes (mostly changes from pointer to array parameters, and `in out` versus `in` and using `cardinals` instead of `naturals`)

`Xt_Get_Resources`	approved
`Xt_Get_Subresources`	approved
`Xt_Get_Application_Resources`	approved
`Xt_Get_Values`	approved
`Xt_Get_Subvalues (no widget_id arg.)`	approved
`Xt_Set_Subvalues (no widget_id arg.)`	approved
`Xt_Set_Values`	approved

68 add `Xt_Get_Resource_Offset` to semiprivate; unique to Unisys (Mark Nelson will examine usability for SAIC) — approved

69	make changes to `set_resource` after interface decided	deferred
70	add semiprivate package to `resource_management`; check if pointers should be moved to semiprivate and procedures to body	deferred
	delete the following; should be in `Xt_Convert` package body or in semiprivate	deferred
71	`Xt_Set_Default_Converter_Table`	
72	`Xt_Free_Converter_Table`	
73	`Xt_Table_Add_Converter`	
74	delete `Xt_App_Converter`; X11R2-only procedure	approved
75	`Xt_Convert_Sub_Program` specification changes (mostly changes from pointers to array parameters and `in out` versus `in`)	approved

	`Xt_App_Add_Converter`
	`Xt_Direct_Convert`
	`Xt_Convert`	

76	delete `Xt_Convert` (with app parameter) procedure; should be in `Xt_Convert` package body instead	approved
77	move `Xt_Event_Initialize`, `Xt_Register_Window`, and `Xt_Unregister_Window` to `event_management` semiprivate	approved
78	move these objects to body of `event_management`:	approved

	`table`
	`grab_list`
	`free_grabs`
	`focus_list`
	`Xt_Destroy_List`
	`only_keyboard_grabs`
	`focus_trace_good`
	`expose_region`	

```
app_destroy_list
Xt_App_Destroy_Count
Xt_Dpy_Destroy_List
```

79 move `Xt_Compile_Callback_List` procedure to `Xt_Callbacks` package semiprivate — approved

80 delete the following; should be in `Xt_Intrinsics` body — approved

```
call_class_part_init
class_init
call_initialize
call_constraint_initialize
```

81 move subprograms to `Xt_Instance_Management`: — approved

```
Xt_Create
Xt_Create_Widget
Xt_Create_Window
Xt_App_Create_Shell
Xt_Realize_Widget
Xt_Unrealize_Widget
Xt_Destroy_Widget
```

82 delete `Xt_Create_Application_Shell` function; for X11R2 - compatibility only; not needed — approved

83 delete `Xt_Phase_2_Destroy` callback object; should be in `instance_management` body — approved

84 change `Xt_Callback_List` to private type — approved

85 add `Xt_Callback_Struct` to `Xt_Callbacks` package — approved

86 `Xt_Callbacks` subprogram declaration changes (**`in out`**) — approved

```
Xt_Add_Callback
Xt_Remove_Callback
Xt_Add_Callbacks
Xt_Remove_Callbacks
Xt_Call_Callbacks
```

87 add `Xt_Callbacks` semiprivate package — approved

88 add private section to Xt_Callbacks — approved

89 move Xt_Set_Arg and Xt_Merge_Arg_Lists to Xt_Resource_Management, with arrays passed for the command line arguments (see the earlier RM discussion) — approved

90 delete Xt_Add_Actions; X11R2 compatibility function — approved

91 move Xt_Bind_Actions to translation_management — approved

92 move Xt_Parse_Accelerator_Table, Xt_Install_All_Accelerators, and Xt_Set_Key_Translator to translation_management — approved

93 change Xt_Free(Xt_Popdown) to an actual function because of private widget definition. — approved

This Xt_Free procedure, and many of the other overloaded Xt_Frees, will do an UNCHECKED_DEALLOCATION of both the record being passed in and any private data allocated within that record.

94 declaration change for Xt_Create_Popup_Shell function in Xt_Popups package — approved

95 delete menu_popup and menu_popdown procedures; need not be visible; added to action list at app_context initialization — approved

96 add Xt_Popups semiprivate (with Xt_Popup procedure) — approved

97 Xt_Instance_Management subprogram declaration changes:

Xt_Display_Initialize	deferred, check if argc need
Xt_Open_Display	approved
Xt_Create (moved to semiprivate)	approved
Xt_Manage_Children	approved

98 add Xt_Create_Managed_Widget function to Xt_Instance_Management specification — approved

99 delete Xt_Create_Application_Shell; X11R2 only — approved

100 add Xt_Instance_Management semi_private — approved

101	overload Xt_Create in semiprivate to provide Unisys and SAIC semiprivate functions	make single operation
102	move errordb and error_file objects to environment specific package because of filename path	approved
103	move Xt_Error_Msg_Handler_Procs package with intrinsics procedure pointer routines	approved
104	remove Xt_<no app_>* routines; X11R2-only Xt_Warning error handling functions	keep and Xt_Error

Xt_Warning and Xt_Error will use a default application context since a user defined context might not have been created by the time one of these error handling subprograms are called. This default context is hidden in the package body and is used as a last resort.

105	delete Xt_Init_Error_Handling procedures; unless these needed in semi-private	deferred to check it out
106	Xt_Error_Management subprogram declaration changes: Xt_App_Error_Message (change to msg, remove params parameter) Xt_App_Warning_Message (ditto)	approved
107	delete procedure pointer packages in selection; leave in Xt_Selection_Management package body: Xt_Cancel_Convert_Selection_Procs Xt_Convert_Selection_Incr_Procs Xt_Cancel_Selection_Callback_Procs	approved
108	move procedure pointers with rest of procedure pointer packages: Xt_Selection_Callback_Procs Xt_Close_Selection_Procs Xt_Selection_Done_Procs Xt_Convert_Selection_Procs	approved
109	delete Xt_Set_Selection_Timeout and Xt_Get_Selection_Timeout; X11R2 only subprograms	approved
110	add Xt_Selection_Management semiprivate	approved

111	delete `Xt_Get_Cursor`, not part of Xt	approved
112	move package `utilities` functions to intrinsics; delete `utilities`	approved
113	intrinsics subprogram declaration changes: `Xt_Set_Sensitive` (`widget_id` is now **`in out`** parameter) `Xt_Set_Mapped_When_Managed` (function returns `Xt_Boolean` value)	approved
114	add `Xt_Class_Is_Subclass` function; was semiprivate in `utilities`; now fully public because of the widgets call this	approved (see issue #112)
115	delete convenience routines `Xt_To_Address`, `Xt_To_X_Long_Integer`, and `Xt_Free` for `actions_pointer` as these are used only by RM	approved
116	delete `core_part` and `core_class_part` type declarations, moved elsewhere	approved
117	add new types to `Xt_Intrinsics` private part	approved

GLOSSARY

Ada/X Ada binding and/or implementation of the X Window System
Ada/Xlib Ada binding or implementation to Xlib
Ada/Xt Ada binding or implementation to the Xt intrinsics toolkit
ACM Association for Computing Machinery
ADT Abstract Data Type
Anna Annotated Ada
ANSI American National Standards Institute
API Applications Programming Interface
APSE Ada Programming Support Environment

C C programming language
CASE Computer-Aided Software Engineering
CM Configuration Management
COTS Commercial Off-The-Shelf
C/X C binding/implementation of the X Window System
C/Xlib C binding/implementation of Xlib
C/Xt C binding/implementation of Xt

DEC Digital Equipment Corporation
DIANA Descriptive Intermediate Attributed Notation for Ada
DQL DIANA Query Language

FIPS Federal Information Processing Standard
FTP File Transfer Protocol

GRAMMI Generated Reusable Ada Man-Machine Interface

HP Hewlett-Packard

IEEE Institute of Electrical and Electronic Engineers

JPL Jet Propulsion Laboratory

LIL Library Interconnect Language
LRM Language Reference Manual

NASA National Aeronautics and Space Administration
NIST National Institute of Science and Technology

OL Open Look
OOD Object-Oriented Design
OOP Object-Oriented Programming
OSF Open Software Foundation
OSI Open Systems Interconnection

PCB	Procedure Control Block
PEX	PHIGS Extensions to X
PHIGS	Programmer's Hierarchical Interface for Graphics Systems
POSIX	Portable Operating System Interface to UNIX
RM	Resource Manager
SAIC	Scientific Applications International Corporation
SEI	Software Engineering Institute
SIGAda	Special Interest Group on Ada
SPC	Software Productivity Consortium
STARS	Software Technology for Adaptable and Reliable Systems
TAE+	Transportable Application Environment Plus
TCB	Task Control Block
TIM	Technical Interchange Meeting
UI	UNIX International
UIMS	User Interface Management System
VADS	Verdix Ada Development System
VAPI	Virtual Application Programming Interface
VEX	Video Extensions to X
X	X Window System
Xlib	X library
Xt	X toolkit

INDEX

Ada 9X 31, 32
Ada/X 1, 2, 7, 8, 9, 11, 12, 13, 15, 18, 20, 26, 28, 31, 32, 33, 35, 36, 37, 39, 40, 56, 57, 58, 59, 66
Ada/Xlib 7, 9, 11, 12, 13, 15, 31, 33, 34, 35, 37, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 64, 65, 69, 70, 71, 72, 73, 75
Ada/Xt 11, 15, 16, 20, 21, 23, 26, 27, 28, 29, 31, 33, 35, 37, 46, 49, 52, 56, 57, 58, 59, 60, 63, 64, 65, 66, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77
ADT 24, 25, 46, 51, 77
Alsys 11, 45
ANSI 33, 34, 39
API 2, 6, 7, 35
application developer 2, 5, 6, 15, 16, 18, 20, 22, 23, 26, 28, 29, 31, 32, 34, 35, 36, 37, 39, 43, 49, 56, 57, 58, 59, 64, 65, 70, 71, 72, 73, 74
APSE 36
Athena 5, 6, 7, 9, 15, 31, 34, 35, 37, 39, 43, 58, 73
awk 58, 65, 72
binding developer 1, 2, 7, 12, 13, 15, 21, 23, 25, 26, 27, 28, 29, 31, 32, 33, 35, 36, 37, 39, 43, 44, 46, 48, 49, 51, 52, 55, 56, 57, 58, 59, 60, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 77, 78

C/X 9, 31, 33
C/Xlib 12, 15, 43, 45, 46, 47, 48, 50, 51, 52, 70
C/Xt 15, 20, 23, 26, 27, 28, 46, 56, 57, 59, 60, 63, 64, 65, 66, 68, 69, 70, 71, 74, 75
call_data 56
CASE 2
cc 50
Change_Geometry 60
closure 56
CM 11
command 29
Command_Line_Arguments 65
composite 21, 29
composite_class 62
composite_rec 62
Configuration_Dependent 45
constraint 21
constraint_class_part 63
constraint_widget_rec 63
context 44
core 21, 23, 25, 29, 69
core_class_part 61
Core_Part 61
COTS 1, 36, 39
create_widget 72

DECAda 11, 50
DECWindows 8
devices 75
DIANA 20
DQL 20
drawable 44

emacs 43
error_handler 77
error_proc 77
Events 46
execute 56, 57
extension 28

FIPS 158 35, 36
framework 7
FTP 37
func 56
func_ptr 56, 57

gadgets 23
gcc 50
get_resources 27
GHG 7, 8, 37
GRAMMI 8
grebyn 37

handle 56
hash_bucket 48
HP 7, 8

IEEE 33, 34, 39, 50
inherit 57

intrinsics 5, 7, 15, 16, 21, 23, 28, 29, 37, 39, 56, 58, 59, 60, 64, 65, 69, 71, 72, 74

JPL 8

Karlsruhue 53
key_code 52

label 25, 29
LIL 57
local_ptr 56

malloc 28, 60
Mask_Type 44
Meridian 11, 50, 68
Motif 6, 8, 21, 31, 34, 35, 36, 37, 58, 59, 66, 70
NASA 2, 8
NIST 35, 36
null_address 52

object 64
object_class_part 63
object_widget_part 63
OL 6, 21, 31, 34, 35, 36, 58, 59, 66, 70
OOP 28
OSF 6, 7, 8, 34, 60, 70
OSI 35

P1003 36
.5 34, 49, 51, 74
P1201 33
.1 34
.3 35
.4 34, 35
.5 35
PCB 56, 57, 58
PEX 36
PHIGS 36
point 52
POSIX 33, 34, 35, 36, 49, 51, 74
Posix_Strings 51
proc_id 56, 57
proc_ptr 56, 57, 65
quark 50, 69
quark_array 50, 51
quark_list 50

R1000 9
Rational 9, 34
rectangle 52
reference model 7, 36
Renamed_Xlib_Types 65
Resource Manager (RM) 23
resource_mgr 47, 48
resource_records 27
RM 23, 25, 26, 27, 28, 46, 47, 59, 68, 69, 72

Sanders 9
screen_list 53
search_lists 48
sed 43, 49
SEI 2, 8
Serpent 2, 8, 35
set_arg 69
set_resource 26
set_resources 27
shell 21, 29, 57, 58, 59, 64
SIGAda 31, 33
simple 25
SPC 8
STARS Foundation 7
STARS Foundation's 7, 13, 37
STARS Foundations 8, 35, 43, 50, 51
strip 70
Subprogram_Pointer 56, 57, 58, 60, 64, 65, 66, 69
Systeam 58
System_Environment 45
System_Utilities 45

TAE+ 2, 8, 35
TCB 73
TeleSoft 8, 11
TeleUSE 8
the_obj 56
To_Xt_Argval 26
translation_management 78
TRW 8

UI 6, 34, 60
UIMS 2, 6, 8, 32, 33, 35, 36, 39
User_Supplied 56, 57

VADS 11
VAPI 34

Verdix 11, 50
VEX 36

widget developer 2, 6, 7, 13, 15, 16, 18, 20, 21, 22, 23, 24, 25, 26, 27, 29, 31, 39, 56, 57, 58, 59, 64, 65, 66, 70, 71
widget_id 56
widget_resource_ptr 28
widget_resources 28
window_list 53

X protocol 5, 6, 9, 34, 36, 45
x-ada 37
X11R2 7, 35, 60, 66
X11R3 15, 31, 35, 36, 39, 49, 51, 53, 60, 69
X11R4 7, 12, 15, 28, 31, 35, 36, 39, 49, 51, 69, 76
X11R5 7, 9, 28, 31, 34, 35, 39, 76
X11R6 31, 35, 74
X3H3 33
.6 34
X_Address 52
X_Id 44
X_Lib 43, 45, 49, 64, 71
X_Lib_Interface 11, 53
X_Windows 43
Xlib 5, 7, 8, 9, 11, 12, 33, 34, 35, 36, 43, 45, 46, 47, 48, 49, 50, 51, 52, 64, 70
Xmu 12
Xray 7, 35
Xrm 48
XrmDatabase 53
XrmDestroyDatabase 53
Xt 5, 7, 8, 15, 16, 20, 21, 23, 28, 31, 33, 34, 35, 36, 37, 46, 56, 58, 60, 63, 64, 71, 72, 74
Xt_Anciallary_Types 65
Xt_App_Create_Shell 28, 59, 60
Xt_Boolean 27, 28, 65
Xt_Box_Widget 72
Xt_Box_Widget_Class 72
Xt_Callbacks 67
Xt_Core 64
Xt_Create_Managed_Widget 28, 59
Xt_Error 73, 84
Xt_Error_Management 68
Xt_Error_Raised 75
Xt_Event_Management 66
Xt_Exit 73, 74, 75
Xt_Free 65, 83
Xt_Geometry_Management 67
Xt_Get_Value 26
Xt_Get_Values 27
Xt_Instance_Management 68
Xt_Intrinics 65, 66
Xt_Intrinsics 23, 28, 60, 64, 77
Xt_Label 26
Xt_List_Overflow 76
Xt_Popups 67
Xt_Procedure_Types 65, 66
Xt_Resources 66, 68, 69
Xt_Selection_Management 67
Xt_Set_Arg 26
Xt_Set_Values 27
Xt_Translation_Management 67
Xt_Utilities 68, 75
Xt_Warning 84
XtCore 21
XVT 34
XWithdrawWindow 52

York 58

zero_address 48
Zero_X_Address 52